MÉTODO DE PREDICCIÓN AUTOMÁTICO PARA EVALUAR LOS ATRIBUTOS DE CALIDAD EN PATRONES DE DISEÑO BASADOS EN PROGRAMACIÓN ORIENTADA A OBJETOS DURANTE LA ETAPA DE DESARROLLO DE APLICACIONES DE SOFTWARE

JUAN PABLO DUQUE ALFONSO

FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE SISTEMAS
TRABAJO DE INVESTIGACIÓN TECNOLÓGICA
UNIVERSIDAD CATÓLICA DE COLOMBIA
Bogotá, 2018

MÉTODO DE PREDICCIÓN AUTOMÁTICO PARA EVALUAR LOS ATRIBUTOS DE CALIDAD EN PATRONES DE DISEÑO BASADOS EN PROGRAMACIÓN ORIENTADA A OBJETOS DURANTE LA ETAPA DE DESARROLLO DE APLICACIONES DE SOFTWARE

JUAN PABLO DUQUE ALFONSO

Trabajo de Grado para optar al título de
INGENIERO DE SISTEMAS

Director
ROGER GUZMÁN
M. Sc. Ingeniería de Sistemas y Computación

FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE SISTEMAS
TRABAJO DE INVESTIGACIÓN TECNOLÓGICA

UNIVERSIDAD CATÓLICA DE COLOMBIA
Bogotá, 2018

Nota de Aceptación

Aprobado por el comité de grado en cumplimiento de los requisitos exigidos por la Facultad de Ingeniería y la Universidad Católica de Colombia para optar al título de ingeniero de sistemas.

_____

Nombres y Apellidos del director
Director

_____

Jurado 1

_____

Jurado 2

_____

Revisor Metodológico

Bogotá. 17, mayo, 2019.

DEDICATORIA

DEDICATORIA
Este trabajo lo dedico a mis hermanas y mi madre.

AGRADECIMIENTOS

Agradezco a la Universidad Católica de Colombia y a todo el cuerpo docente por la formación profesional obtenida y por todos los que fueron partícipes en este proceso de inicio a fin, quienes me brindaron las herramientas necesarias para ser un gran profesional, particularmente el Ingeniero Roger Enrique Guzmán, quien hizo parte de este proyecto a través de su conocimiento, tiempo y dedicación.

RESUMEN

Este trabajo pretende llevar a cabo el análisis de calidad de proyectos de software según sus atributos de calidad en una etapa temprana a través de modelos estadísticos que generen la mayor eficiencia posible. Como parte de su ejecución, se hace uso de herramientas estadísticas que permitan efectuar la clasificación y predicción de los datos más relevantes (según sea el caso), con el uso de estas herramientas permite detectar de forma eficaz los potenciales problemas que se llegasen a presentar, permitiendo así una retroalimentación eficiente.

Para lograr este cometido se crea Lugh, el cual es el resultado de este trabajo, para su desarrollo se utiliza el API de Git, el cual obtiene código fuente del proyecto que se desea analizar, una vez obtenido el código fuente, se procede a realizar la detección de patrones de diseño y sobre estos se realiza el análisis de código estático, el cual genera un reporte de los errores presentados durante el desarrollo del proyecto, finalmente mediante el uso de procesos estadísticos, como Naive Bayes, se logra la predicción de la cantidad de errores en el código, o en otras palabras se puede predecir la cantidad de errores en un proyecto de software hasta cuando el código del proyecto analizado sea publicado .

Los resultados obtenidos del proyecto se encuentran en repositorios de Git sobre los cuales se realizaron pruebas y predicción de errores, dada la medición de precisión de la predicción se obtiene un rendimiento mayor al 90% de predicción de errores para el código fuente analizado.

Palabras clave: calidad, software, automatización, clasificación, iteración, API, código, análisis, predicción, error.

TABLA DE CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE FORMULAS

1. GLOSARIO

Aprendizaje automático: Los algoritmos avanzados de aprendizaje automático están compuestos de muchas tecnologías, que se utilizan en el aprendizaje supervisado y sin supervisión, que funcionan de forma guiada por las lecciones de la información existente.

Arquitectura: En ingeniería de software, es el diseño general de un sistema informático y las interrelaciones lógicas y físicas entre sus componentes. La arquitectura especifica el hardware, software, métodos de acceso y protocolos utilizados en todo el sistema.

Big Data: Big data son activos de información de gran volumen, alta velocidad y / o gran variedad que demandan formas de procesamiento de información innovadoras y rentables que permiten una visión mejorada, toma de decisiones y automatización de procesos.

Bug: Es un problema inesperado con el software o hardware. Los problemas típicos suelen ser el resultado de una interferencia externa con el rendimiento del programa que el desarrollador no anticipó.

Estándar: Es documento que recomienda un protocolo, interfaz, tipo de cableado o algún otro aspecto de un sistema. Incluso puede recomendar algo tan general como un marco conceptual o modelo

IDE: IDE (entorno de desarrollo integrado) Entornos para escribir lógica de aplicaciones y diseñar interfaces de aplicaciones.

Integración Continua: Los sistemas de integración continua (CI) proporcionan la automatización del proceso de compilación y validación del software de forma continua mediante la ejecución de una secuencia de operaciones configurada cada vez que un cambio de software se verifica en el repositorio de administración de código fuente.

Inteligencia Artificial: La inteligencia artificial (IA) aplica análisis avanzados y técnicas basadas en la lógica, incluido el aprendizaje automático, para interpretar eventos, respaldar y automatizar decisiones y tomar medidas.

Sistema Experto: Un sistema de software que puede aprender nuevos procedimientos al analizar el resultado de eventos pasados o que contiene una base de conocimientos de reglas que se pueden aplicar a nuevos datos o circunstancias que el desarrollador no haya previsto explícitamente.

2. ACRÓNIMOS

API: Application Programming Interface
CI: Continuous Integration
CD: Continuous Delivery
DP: Design Pattern
GOF: Gang of Four
IA: Inteligencia Artificial
IDE: Integrated Development Environment
INCOSE: International Council on Systems Engineering
MISRA: Motor Industry Software Reliability Association
POSA: Pattern Oriented Software Architecture
QA: Quality Assurance
ROC: Receptor Operative Characteristic
SE: Standard Edition
VPN: Valor Predictivo Negativo
VPP: Valor Predictivo Positivo
XP: Extreme Programming

3. INTRODUCCIÓN

El reto al que actualmente se enfrentan los arquitectos, diseñadores y desarrolladores de software, obedece en gran parte a la necesidad de conocer la eficacia y el buen comportamiento de la aplicación en la que se está trabajando. Debido a la celeridad que se requiere para entregar un proyecto, además del vínculo emocional que tienen los desarrolladores con su creación, la eficacia del software entregado no es la óptima[1], entre otras cosas, porque solo se tienen en cuenta aspectos y escenarios positivos que se pueden integrar al software y adaptarlos al producto que se desea entregar. En concordancia con lo anterior, el proceso de calidad de software es fundamental debido a las características que se deben tener en cuenta como las mencionadas por Maneela Tuteja y Gaurav Dubey en su artículo "*La importancia de las pruebas y Aseguramiento de la calidad en el desarrollo de software*"[2], las cuales hacen referencia a los procesos implementados a lo largo del tiempo y llevados a cabo de forma manual o automatizada mediante procesos que han sido programados por un analista de control de calidad especializado.

[1] International Software Testing Qualifications Board. (2011). Certified Tester Foundation Level Syllabus, 85.

[2] Tuteja, M., & Dubey, G. (2012). A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models, (3), 251–257.

4. PLANTEAMIENTO DEL PROBLEMA

Debido a la prioridad que se le da al valor funcional durante el desarrollo del software siguiendo las reglas del manifiesto Ágil: "*Nuestra máxima prioridad es satisfacer al cliente a través de la entrega temprana y continua de un software funcional.*"[3], se dejan de lado aspectos importantes para tener en cuenta durante dicho proceso, tales como escenarios fallidos y características no funcionales, que ante los ojos del usuario final solo se ven reflejados una vez este falle. Una de las formas de evitar este problema, es mediante el conocido modelo de Gatekeeper o modelo de revisión de pares, donde los desarrolladores del proyecto verifican el código fuente manualmente bajo la premisa de un juicio basado en la experiencia. Sin embargo, no es una revisión profunda, debido a que muchas reglas tienden a ser evitadas, pasadas por alto o asumidas, de tal manera que, al ser revisado por un miembro del equipo, este puede incurrir en errores y el proceso demanda tiempo adicional para llevarse a cabo.

Al respecto conviene mencionar el proceso automático de verificación, *SonarQube Quality Gate*[4] , el cual establece una serie de reglas para evitar problemas en el desarrollo del software, como el *Análisis de Código Estático*. Un ejemplo de ello corresponde a la verificación que este realiza basado en reglas definidas por *Sonar* o por el propio arquitecto. No obstante, una falencia de estas *Quality Gates* es que algunas de las reglas no aplican al software en construcción o podrían carecer de las reglas requeridas para el desarrollo de esta. Es importante aclarar que en la medida que la industria evolucione, también lo deben hacer dichas reglas, requiriendo que el arquitecto, el desarrollador o el diseñador las actualice, conllevando a un proceso más extenso.

Actualmente en los proyectos de desarrollo se afirma que: "*los proyectos ágiles tienen iteraciones cortas que permiten al equipo del proyecto recibir comentarios tempranos y continuos sobre la calidad del producto durante todo el ciclo de vida del desarrollo.*"[5], lo que indica que cuanto antes se realicen las pruebas, menor es el costo de repararlo. Ahora bien, en el proceso de Integración Continua /Despliegue Continuo, todas las pruebas de características no funcionales se ejecutan en el último ambiente también conocido como Servidor de Aceptación de Usuario o Preproducción, el cual, en caso de encontrar un error, regresa todo el proceso a la etapa inicial de desarrollo, haciéndolo costoso e ineficiente.

---

[3] Scrum Values [en línea] scrum-values. [Citado el 24 mayo, 2018] Disponible en internet: <https://www.scrumalliance.org/learn-about-scrum/scrum-values>

[4] Análisis de Código [en línea] Analyzing Source Code. [Citado el 24 mayo, 2018] Disponible en internet: <https://docs.sonarqube.org/display/SONAR/Analyzing+Source+Code>

[5] International Software Testing Qualifications Board. (2011). Certified Tester Foundation Level Syllabus, 85.

Aunque los métodos mencionados anteriormente ayudan a mitigar los errores planteados, dentro de los inconvenientes que se presentan al confiar esta tarea a los integrantes de un equipo, es la demanda de tiempo y recursos.

Por lo tanto, la pregunta de investigación para este proyecto es: ¿Como se pueden mitigar los errores y reducir el costo del proceso de calidad mediante herramientas automatizadas?

5. OBJETIVOS

5.1 OBJETIVO GENERAL

Implementar un modelo computacional de predicción usando aprendizaje de máquina que permita medir los atributos de calidad en patrones de diseño de software basado en programación orientada a objetos.

5.2 OBJETIVOS ESPECÍFICOS

- Construir un conjunto de datos conformado por archivos fuente en lenguaje Java, que cumplan con estándares de patrones de diseño de Software basado en programación orientada a Objetos.

- Desarrollar una estrategia para la categorización y predicción automática del conjunto de datos, a partir de un grupo de patrones de diseño de software basados en programación orientada a objetos.

- Medir el desempeño del modelo predictivo, para evaluar los atributos de calidad de los patrones de diseño de software, basado en programación orientada a objetos usando medidas tales como: exactitud y área bajo la curva.

6. JUSTIFICACIÓN

Al iniciar de manera temprana la ejecución de pruebas dentro de un proceso de desarrollo de software, se reducen costos, tiempo y errores en la entrega del producto al cliente[6]. Hoy en día, técnicamente hablando, el primer proceso para garantizar la calidad del software se lleva a cabo mediante las pruebas unitarias y el análisis estático de código, estas pruebas se ejecutan prácticamente de inmediato una vez que el entregable se considera terminado. Sin embargo, estas pruebas no son completamente confiables y solo miden una pequeña parte de todo el mapa[7].

Incluso las técnicas de predicción de defectos utilizadas en la actualidad se construyen utilizando una gran cantidad de datos históricos. No obstante, en caso de que se construya un nuevo proyecto, no hay registros antecedentes y muchas organizaciones a menudo no conservan dichos datos. En ese caso, se puede proporcionar un modelo inicial para tener un punto de entrada sobre un único módulo, para luego construir un modelo de predicción de defectos que pronostique la propensión de los defectos en otros módulos[8], realizando diversas actividades para garantizar la calidad. Dentro de las que se incluye establecer requisitos y controlar cambios, establecer el método de implementación y lograr la calidad específica del producto y, finalmente, evaluar el proceso y la calidad del producto final.

Para caracterizar el marco de calidad extendida, primero se explican en detalle el conjunto de definiciones y conceptos relacionados. La calidad del producto se detalla mediante el uso de las definiciones de los atributos del producto como base para establecer los requisitos de calidad, los métodos para ayudar a cumplir estos requisitos y la garantía de calidad.[9] Teniendo en cuenta estas definiciones, los datos históricos pueden consolidarse permitiendo, a su vez, analizar el rendimiento de predicción para el conjunto de datos dado, a través de esquemas de evaluación y constructores de predicción de defectos, que de acuerdo con los esquemas de aprendizaje evaluados, sea posible predecir dichos defectos del software con nuevos datos teniendo en cuenta la parametrización del modelo[10].

---

[6] Myers, G. J., Badgett, T., & Sandler, C. (2011). Software Testing Tutorial, 2–7.

[7] Louridas, P. (2006). Static code analysis. IEEE Software, 23(4), 58–61.

[8] Li, M., Zhang, H., Wu, R., & Zhou, Z. H. (2012). Sample-based software defect prediction with active and semi-supervised learning.

[9] Kenett, R., & Baker, E. (2010). Process Improvement and CMMI® for Systems and Software.

[10] Song, Q., Jia, Z., Shepperd, M., Ying, S., & Liu, J. (2011). A general software defect-proneness prediction framework.

7. MARCO DE REFERENCIA

En este capítulo se realiza una revisión bibliográfica de los conceptos involucrados en la estrategia de solución, que permita al lector contextualizar los siguientes capítulos.

Se toma como base el desarrollo y las fases de pruebas utilizadas en la actualidad, seguido del ciclo de CI/CD (Integración Continua) y cómo las fases de pruebas se ven afectadas. Finalmente se abordan los modelos desarrollados con inteligencia artificial para mitigar la propagación de errores en el desarrollo de software.

7.1 MARCO TEÓRICO

7.1.1 Pruebas y mitigación de errores

Durante el desarrollo de las aplicaciones de software es necesario que se tengan en cuenta los procesos de pruebas. En metodologías ágiles como XP y Scrum las fases se han visto alteradas para poder agilizar dicho proceso.

El ciclo de pruebas se enfoca en un modelo de excelencia[11] y estas han ido evolucionando al par con las metodologías ágiles. En el modelo regular, se pueden apreciar modelos considerados arcaicos para lograr la velocidad necesaria que el equipo requiere y poder adaptarse a los cambios que el usuario final pueda necesitar[12]. De los modelos evaluados se toman dos modelos como referencia: el Modelo en Cascada y el Modelo en V, para finalmente abordar el Modelo Ágil en el cual se enfoca el proyecto.

En la figura 1, se encuentra el modelo de mayor antigüedad (Modelo en Cascada). Este modelo lo documentó Benington por primera vez en 1956 y fue modificado por Winston Royce[13] en 1970. Este modelo se enfoca en ejecutar fases estáticas de desarrollo con posibilidades de pasar de una fase a otra sin retroalimentación posterior, donde cada fase del desarrollo es estrictamente para una sola actividad específica, desperdiciando tiempo y recursos, y, como se explica en la justificación de este documento, no hay una fase temprana de identificación de potenciales

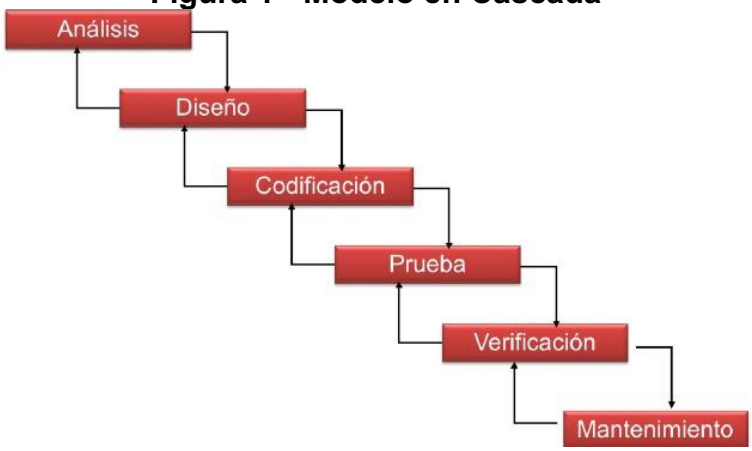
[11] Why agile development races ahead of traditional testing. [en línea] computerweekly. [Citado el 24 mayo, 2018] Disponible en internet: <https://www.computerweekly.com/feature/Why-agile-development-races-ahead-of-traditional-testing>

[12] Balaji, S. (2012). Waterfall vs v-model vs agile : A comparative study on SDLC. WATERFALL Vs V-MODEL Vs AGILE : A COMPARATIVE STUDY ON SDLC, 2(1), 26–30.

[13] Rajagopalan, S. (2014). REVIEW OF THE MYTHS ON ORIGINAL SOFTWARE DEVELOPMENT MODEL, 5(6), 103–111.

problemas conllevando a un elevado costo de recursos y tiempo al tener que ser revaluado, obligando de esta manera al equipo a retomar el proceso desde el principio.

Figura 1 - Modelo en Cascada



Fuente: https://analisisdesistemas1.files.wordpress.com/2015/03/etapa1.jpg

En el Modelo en V, detallado en la figura 2 (el cual fue presentado en el simposio de NCOSE de 1991 en Chattanooga, Tennessee[14]), se puede apreciar un proceso de desarrollo con análisis y una evolución en paralelo de los requerimientos, para llegar a un modelo en el que los recursos están preparados para realizar una entrega rápida del producto final, sin embargo este modelo aún sigue siendo lineal por lo que la retroalimentación y la detección de "bugs" o potenciales problemas se encuentran en una fase muy avanzada del proceso, reiterando el aumento de costos si adicionalmente los potenciales problemas son de carácter técnico, ya que la fase en la cual se evalúan estos aspectos están en el pico más alejado del proceso y sin una actividad paralela que permita retroalimentar los posibles errores.

Estos dos modelos se basan en la premisa mencionada de ejecutar y crear escenarios de forma manual, de manera que con el Modelo Ágil se evidencia un cambio significativo, refiriendo de primera mano la comunicación que se facilita entre los desarrolladores y analistas con mayor naturalidad, cara a cara[15]. Adicionalmente, el paradigma de trabajo manual es altamente reducido debido que las pruebas bajo este modelo son automatizadas, siendo una gran herramienta dentro de los diferentes procesos que se llevan a cabo otorgando más agilidad, permitiendo, además, ser ejecutadas en cualquier ciclo del desarrollo de manera rápida y con un costo mínimo en el desarrollo de esta[16]. Cabe aclarar que estas

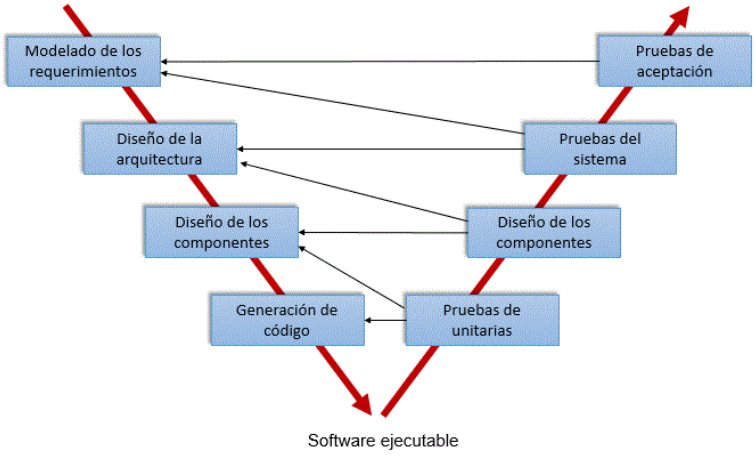
[14] Ruparelia, N. B. (2010). Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3), 8.

[15] Supporting agile manifesto principle 6: face-to-face contact [en línea] Platinum Edge. [Citado el 24 mayo, 2018] Disponible en internet: <https://platinumedge.com/blog/supporting-agile-manifesto-principle-6-face-face-contact>

[16] Dustin, E., Rashka, J., & Paul, J. (2017). Automated software testing, (April), 1–37.

pruebas aún no están enfocadas en aspectos técnicos como el desempeño de la aplicación, estas se encuentran netamente direccionadas en probar la funcionalidad. Para tal objetivo se encuentran otro tipo de pruebas que únicamente se tienen en cuenta en una etapa de desarrollo posterior.

Figura 2 - Modelo en V



Fuente: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro17/modelo_de_cascada.gif

Teniendo en cuenta esta metodología ágil a continuación se explica como este proceso se une con la Integración Continua.

7.1.2 Integración continua

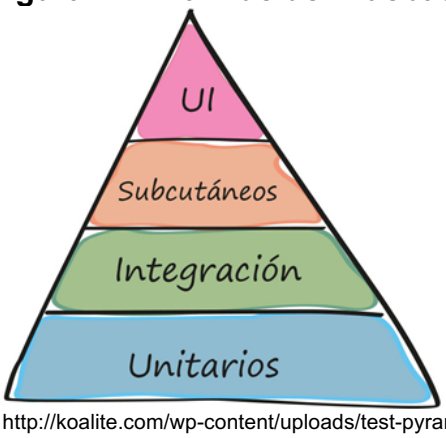
En el modelo de integración continua (CI por sus siglas en ingles), se logra un sistema eficaz que permita construir y probar una aplicación en proceso de desarrollo, las primeras incursiones en el proceso de integración continua fueron dadas por Martin Fowler[17] en el año 2000, su idea no se basaba solamente en la integración de proyectos de desarrollo. Según Fowler su objetivo era la automatización, dando como resultado un proceso exitoso o fallido, con un proceso automatizado que permitiera obtener los archivos fuente, compilar, relacionar, desplegar y ejecutar todo tipo de pruebas automatizadas, teniendo en cuenta la pirámide de pruebas[18] detallada en la Figura 4, dentro de las que se encuentran: pruebas unitarias, pruebas de servicio y pruebas de interfaz gráfica.

[17] Continuous Integration [en línea] Martin Fowler. [Citado el 5 junio, 2018] Disponible en internet: <http://martinfowler.com/articles/originalContinuousIntegration.html>

[18] Practical test pyramid [en línea] Martin Fowler. [Citado el 5 junio, 2018] Disponible en internet: <https://martinfowler.com/articles/practical-test-pyramid.html>

Esta pirámide tiene en cuenta características de cantidad, de escenarios, velocidad de creación y ejecución, niveles de integración y aislamiento. De acuerdo a lo descrito por Mike Cohn[19], en la base se denota gran cantidad de pruebas unitarias, compuestas por pruebas más granulares de todo el ciclo de vida en el desarrollo de software, donde el interés es poder tomar una unidad muy pequeña del código escrito y probar que su funcionalidad sea la esperada, en caso de un lenguaje orientado a objetos, se busca la mínima parte en el código, el cual es un método dentro de una clase; en el caso que sea un lenguaje funcional, se piensa en una función única[20]. Las siguientes fases están pensadas en verificar funcionalidad y la integración de los componentes, en los cuales se pueden encontrar pruebas de integración, pruebas de contrato, pruebas de consumo, pruebas de interfaz de usuario y pruebas de aceptación. Adicionalmente, fuera de la pirámide de pruebas, se llevan a cabo pruebas (valga la redundancia) que actualmente se ejecutan en las ultima fase de desarrollo y no son funcionales[21], en estas se incluyen las pruebas de desempeño y seguridad.

Figura 4 - Pirámide de Pruebas



Fuente: http://koalite.com/wp-content/uploads/test-pyramid.png

Retomando el proceso de CI y teniendo en cuenta lo descrito por la pirámide de pruebas, estas tienen una estrecha relación, debido a que estas pruebas automatizadas son ejecutadas por medio de una herramienta de CI, discriminadas por tipo de prueba, fase de desarrollo y ambiente de ejecución, como se puede ver en la figura 5. En el proceso de integración continua se definen las pruebas a ejecutar y su ambiente, donde las pruebas unitarias, que son las más extensas, granulares y rápidas, se ejecutan en un ambiente de desarrollo teniendo en cuenta

[19] Cohn, M. (2009). The Forgotten Layer of the Test Automation Pyramid.

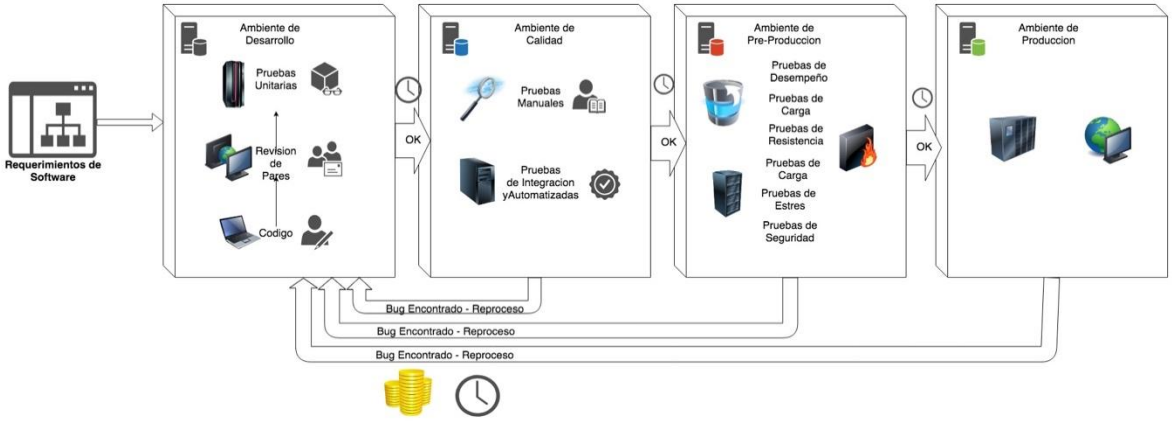
[20] Hauser, T. (2009). the Art of Unit Testing. Library Journal (Vol. 128).

[21] Jeskanen, M. (2015). Non-functional testing: security and performance testing, (November).

el proceso de integración continua, las cuales a su vez se ejecutan en un servidor individual, donde los desarrolladores tendrán la versión del Software en desarrollo más actualizada con los elementos entregados, que de acuerdo a la estrategia de ramas[22], este ambiente tendrá la rama *Feature*, en la cual los cambios de todos los desarrolladores están integrados(ver la figura 5), y en ella se puede observar el comportamiento de las ramas. En la siguiente fase se encuentran las pruebas funcionales automatizadas y de integración, la cual se ejecuta en un ambiente de calidad o de *QA*, y la rama que ejecuta estas pruebas es *Dev*.

Finalmente se encuentra el ambiente de preproducción, dentro del cual todas las pruebas no funcionales son ejecutadas en la rama *Release*, en este modelo de *Integración Continua, Metodología Ágil* y *Pruebas Ágiles*, se encuentra un proceso totalmente automatizado con poca intervención humana, permitiendo detectar y mitigar algunos de los problemas potenciales en la entrega final del producto.

Figura 5 - Proceso de Integración Continua



Fuente: Autor

Seguido a esto, algunas investigaciones hechas previamente con Inteligencia Artificial han permitido mitigar en gran medida potenciales errores o fallas en el proceso de desarrollo, que en su mayoría corresponde a características no funcionales.

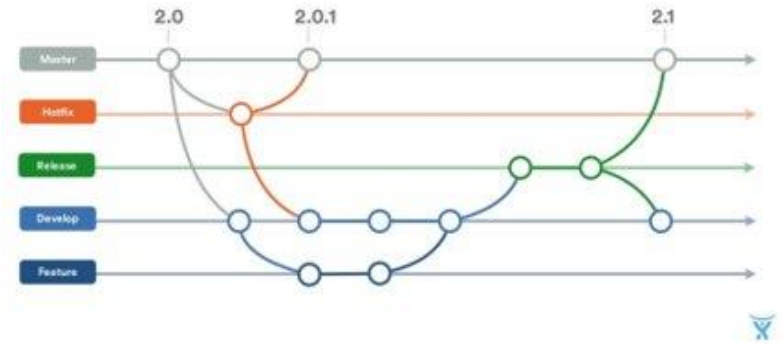
7.1.3 Flujo de trabajo GitFlow

[22] Branching strategies with tfvc [en línea] microsoft. [Citado el 18 febrero, 2019] Disponible en internet: <https://docs.microsoft.com/en-us/vsts/tfvc/branching-strategies-with-tfvc?view=vsts>

El modelo de GitFlow es un flujo propuesto por Atlassian[23], el cual modela el proceso mediante el cual un equipo de desarrollo controla el proceso de la creación de la aplicación, con herramientas de control de versiones como Git o Bitbucket, estas herramientas permiten registrar los cambios históricos del código fuente, versiones y características incluidas en el software, adicionalmente permite publicar a todos los miembros del equipo algún cambio realizado en el proyecto.

El objetivo de este modelo es dividir los cambios que se han realizado a través del tiempo en cinco principales tipos de Ramas: *Máster*, *Release*, *Develop, Feature* y *Hotfix*, siendo flexible en incluir más tipos de ramas como se evidencia en la figura 3, dichas ramas muestran los cambios realizados en un determinado tiempo.

Figura 3 - GitFlow



Fuente: https://i.ytimg.com/vi/gLWSJXBbJuE/maxresdefault.jpg

Las ramas descritas tienen las siguientes funciones:

- **Master**: Esta es una rama altamente estable que siempre está lista para la producción y contiene la última versión de lanzamiento del código fuente en producción.

- **Develop**: Derivado de la rama Master, la rama de Develop sirve como una rama para integrar diferentes características planificadas para una próxima versión. Esta rama puede o no, ser tan estable como la rama Master. Es donde los desarrolladores colaboran y fusionan ramas de características.

- **Feature**: Cada nueva funcionalidad debe residir en su propia rama, que puede enviarse al repositorio central para la copia de seguridad / colaboración. Pero, en lugar de derivarse del Master, las ramas Feature usan Develop como su rama principal. Cuando una característica se completa, se fusiona de nuevo en el Develop. Las características nunca deben interactuar directamente con la rama Master.

[23] A successful Git branching model [en línea] Vincent Driessen. [Citado el 16 abril, 2019] Disponible en internet: < https://nvie.com/posts/a-successful-git-branching-model/ >

- **Release**: Una vez que el desarrollo ha adquirido suficientes características para el despliegue a producción, se puede crear una rama de Release desde la rama de Develop. La creación de esta rama inicia el siguiente ciclo de vida, por lo que no se pueden agregar nuevas funcionalidades después de este punto; solo las correcciones de errores, la generación de documentación y otras tareas orientadas a la liberación, deben incluirse en esta rama. Una vez que está listo para el despliegue, la rama de Release se fusiona con la rama Master y se etiqueta con un número de versión. Además, debe volver a fusionarse con la rama de Develop, que puede haber progresado desde que se inició el despliegue.

- **Hotfix**: Las ramas de Hotfix o "revisión" se utilizan para crear un parche sobre las versiones que se encuentran en producción, el cual debe ser desplegado rápidamente. Las ramas de Hotfix son similares a las ramas de Release y de Feature, excepto que estas son creadas a partir de la rama Master en lugar de Develop. Esta es la única rama que debe crearse directamente de Master. Tan pronto como se complete la corrección, se debe fusionar tanto con la rama de Master como con la rama de Desarrollo y la rama Master se debe etiquetar con un número de versión actualizado.

7.1.4 Modelos en inteligencia artificial

Dentro de los modelos previos de IA, se destaca el propuesto por Li y Henry[24]. En este punto es importante resaltar el papel que desempeña la métrica en el desarrollo del software, debido a la contribución que ejerce en la evaluación de las técnicas empleadas y del control en el proceso, permitiendo llevar a cabo una estimación de los atributos que hacen parte del software. Por consiguiente, las métricas conocidas como *Atributos Externos* y *Atributos Internos* se tuvieron en cuenta en la elaboración del presente documento, debido a la medición que se efectúa en el proceso y los efectos de este con su entorno, respectivamente.

De igual manera, los atributos internos contemplan los impactos de cambio, la propensión a fallas y reusabilidad, incluyendo, además, características tales como herencia, acoplamiento, cohesión, complejidad y tamaño, razón por la cual se tomaron tres alternativas con diferentes algoritmos. La primera alternativa, es una inducción por árboles de decisión representado por *J48*, el cual a su vez implementa el algoritmo *C4.5*[25] siendo este, un algoritmo de aprendizaje supervisado que introduce un modelo de clasificación. La segunda alternativa es una inducción por

[24] Dagpinar, M., Jahnke, J. H., & Canada, B. C. (2003). Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison

[25] Dai, W., & Ji, W. (2014). A MapReduce Implementation of C4 . 5 Decision Tree Algorithm, *7*(1), 49–60.

reglas, usando algoritmos como *Jrip, PART y CN2*. Finalmente se experimentó con *NBTree* que combina clasificadores Bayesianos y clasificadores basados en árboles de decisión, ejecutados sobre programas pequeños en *JAVA.* El resultado obtenido entre estos métodos tiene una exactitud de un mínimo del 60% (usando el algoritmo de NBTree) y un máximo del 89% (usando el algoritmo J48).

Otro de los modelos a destacar es el descrito por Li, Zhang y Wu[26], en el que se describe el uso de algoritmos de Inteligencia Artificial basados en muestras, los cuales se clasifican en tres métodos. El primero consiste en generar muestras con máquinas de aprendizaje, su objetivo es generar muestras clasificándolas como *defectuosas* o *no defectuosas* dentro de un módulo pequeño en la aplicación. Una vez definidos estos ejemplos mediante algoritmos de clasificación como Regresión Logística, Árboles de Decisión o Naive Bayes, se da un proceso de aprendizaje, el cual podrá extenderse a los demás módulos de la aplicación de los cuales no habrá muestras, para que de esta forma se clasifiquen como *defectuosas* o *no defectuosas* manteniendo así los datos de aprendizaje pequeños y poder evaluar los demás módulos con mayor disponibilidad de tiempo y recursos.

El segundo método es la generación de muestras mediante aprendizaje semi-supervisado, en el que el sistema de aprendizaje se da por un pequeño conjunto de pruebas, que permita al sistema analizar módulos y ser capaz de clasificarlos. Para este modelo se aplica el algoritmo CoForest[27] el cual consiste en clasificar los datos de ejemplo como los más relevantes y por medio del algoritmo de Random Forest[28] a través de clasificación de datos de manera aleatoria se reúne la información faltante. El último método consiste en el muestreo con aprendizaje semi-supervisado activo, el cual toma como base el sistema de CoForest y se propone una mejora al mismo, conocido como ACoForest en el cual reúnen toda la información que se tomó aleatoriamente, se organiza en árboles y mediante el aprendizaje activo, los datos son clasificados correctamente.

7.2 MARCO CONCEPTUAL

Este capítulo se centra en definir y conceptualizar los algoritmos de clasificación y de toma de decisiones. Adicionalmente, las reglas que se deben tener en cuenta a la hora de plantear el modelo de predicción que estén acorde a las mejores prácticas.

Los temas para abordar son los patrones y anti-patrones y las mejores prácticas en el desarrollo de software, que permiten predecir el comportamiento basado en las

[26] Li, M., Zhang, H., Wu, R., & Zhou, Z. H. (2012). Sample-based software defect prediction with active and semi-supervised learning.

[27] Li, M., Li, H., Zhou, Z.H(2009). Semi-supervised document retrieval.

[28] Breiman, L.: Random forests. Mach. Learn. 45(1), 5–32 (2001)

reglas establecidas. Seguido a esto, los lineamientos para tener en cuenta de manera computacional para poder abordar los datos de ejemplo o muestras necesarias y finalmente se hace referencia a los algoritmos de clasificación, de toma de decisiones y predicción que son usados dentro del experimento.

7.2.1 Patrones y anti-patrones

"Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software."[29]

Dicho de otra forma, un patrón de diseño permite tener una estructura específica y ya definida de una arquitectura planteada previamente, que permite dar solución a problemas específicos en los cuales se debe tener en cuenta cuando aplicar estos modelos, la descripción del problema a solucionar y las consecuencias de usarlos. A continuación, se explican los patrones dados por POSA (Pattern-Oriented Software Architecture), GOF (Gang of Four)[30] y Design Patterns[31].

7.2.1.1 Patrones creacionales

En ingeniería de software, los patrones de diseño creacional son patrones de diseño que se ocupan de los mecanismos de creación de objetos, tratando de crear objetos de una manera adecuada a la situación. La forma básica de creación de objetos podría ocasionar problemas de diseño o complejidad añadida al diseño. Los patrones de diseño creacional resuelven este problema controlando de algún modo la creación de este objeto. Entre los patrones de diseño creacionales que se pueden encontrar en GOF son:

- **Abstract Factory**
  Crea una instancia de varias familias de clases

- **Constructor**
  Separa la construcción del objeto de su representación

- **Factory Method**
  Crea una instancia de varias clases derivadas

- **Object Pool**
  Evita la adquisición costosa y la liberación de recursos al reciclar objetos que ya no están en uso

[29] ¿Qué es un Patrón de Diseño? [en línea] [Citado el 24 mayo, 2018] Disponible en internet: <https://msdn.microsoft.com/es-es/library/bb972240.aspx>

[30] From, E. (2007). Design Patterns, 368–386.

[31] Bayes, R. T., Laplace, P., & Jeffreys, S. H. (2017). Bayes theorem, 1–11.

- **Prototype**
  Una instancia completamente inicializada para ser copiada o clonada

- **Singleton**
  Una clase de la cual solo una instancia puede existir

7.2.1.2 Patrones estructurales

En ingeniería de software, los patrones de diseño estructural son patrones de diseño que facilitan la identificación de las relaciones entre identidades de manera simple. Entre los patrones de diseño estructurales que se pueden encontrar en GOF son:

- **Adapter**
  Une interfaces de diferentes clases

- **Bridge**
  Separa la interfaz de un objeto de su implementación

- **Composite**
  Una estructura de árbol de objetos simples y compuestos

- **Decorator**
  Agrega responsabilidades a los objetos dinámicamente

- **Facade**
  Una sola clase que representa un subsistema completo

- **Flyweight**
  Una instancia refinada utilizada para compartir de forma eficiente

- **Private Data Class**
  Restringe el acceso de mutadores

- **Proxy**
  Un objeto que representa otro objeto

7.2.1.3 Patrones de comportamiento

En ingeniería de software, los patrones de diseño de comportamiento identifican los patrones comunes de comunicación entre objetos y los crea. Al hacerlo, estos patrones aumentan la flexibilidad al llevar a cabo esta comunicación. Entre los que se pueden encontrar en GOF son:

- **Responsibility Chain**

Una forma de pasar una solicitud entre una cadena de objetos

- **Command**

Encapsula una solicitud de comando como un objeto

- **Interpreter**

Una forma de incluir elementos de lenguaje en un programa

- **Iterator**

Accede de forma secuencial a los elementos de una colección

- **Mediator**

Define comunicación simplificada entre clases

- **Memento**

Captura y restablece el estado interno de un objeto

- **Null Object**

Diseñado para actuar como valor predeterminado de un objeto

- **Observator**

Una forma de notificar el cambio a varias clases

- **State**

Altera el comportamiento de un objeto cuando su estado cambia

- **Strategy**

Encapsula un algoritmo dentro de una clase

- **Template Method**

Transfiere los pasos exactos de un algoritmo a una subclase

- **Visitor**

Define una nueva operación para una clase sin cambios

7.2.1.4 Anti-patrones

Los Anti-patrones, al igual que sus contrapartes los patrones de diseño, definen un vocabulario de la industria para los procesos defectuosos comunes y las implementaciones dentro de las organizaciones. Un vocabulario de nivel superior simplifica la comunicación entre los profesionales del software y permite una descripción concisa de los conceptos de nivel superior.

Un Anti-patrón es una forma literaria que describe una solución común a un problema que genera consecuencias decididamente negativas. El Anti-patrón puede ser el resultado de la incomprensión de un desarrollador a la hora de aplicar estas reglas, quien puede no llegar a tener suficiente conocimiento o experiencia para resolver un problema particular o que ha aplicado un patrón perfectamente bueno en el contexto incorrecto.

Los Anti-patrones proporcionan experiencia en el mundo real en el reconocimiento de problemas recurrentes en la industria del software, proporcionando una solución detallada a los problemas más comunes, adicionalmente proporcionan las herramientas para permitirle reconocer estos inconvenientes y determinar sus causas subyacentes.

Así mismo, los Anti-patrones presentan un plan detallado para revertir estas causas subyacentes e implementar soluciones productivas. Describe eficazmente las medidas que se pueden tomar en varios niveles para mejorar el desarrollo de aplicaciones, el diseño de sistemas de software y la gestión eficaz de proyectos de software.

7.2.2 Análisis de código estático

El análisis de código estático es un método de depuración que permite examinar el código fuente antes de ejecutar un programa. Se realiza a través del análisis de un conjunto de códigos contra una serie de reglas establecidas para el desarrollo de aplicaciones de software[32], este tipo de análisis aborda las debilidades en el código fuente que pueden conducir a vulnerabilidades. Generalmente son ejecutadas por herramientas de análisis de código estático, las cuales intentan resaltar posibles vulnerabilidades dentro del código fuente que no se encuentra en ejecución, mediante el uso de técnicas como el análisis de corrupción y el análisis de flujo de datos. Se usa comúnmente para cumplir con las pautas de codificación, como MISRA[33].

Las diferentes técnicas usadas para el análisis de código estático incluyen el Análisis de flujo de datos, el cual es usado para recolectar los datos del flujo de la información durante una ejecución estática, un ejemplo de ello es el flujo de datos en pruebas unitarias; otra técnica es Grafo de control de flujo (CFG por sus siglas en ingles), el cual realiza una representación gráfica abstracta del software, mediante el uso de nodos que representan bloques básicos. Un nodo, en un grafo representa un bloque; las aristas dirigidas se utilizan para representar las

[32] Static Code Analysis [en línea] OWASP [Citado el 21 marzo, 2019] Disponible en internet: <https://www.owasp.org/index.php/Static_Code_Analysis>

[33] MISRA C : 2012 Technical Corrigendum 1 Technical clarification of British Library Cataloguing in Publication Data. (2017), (June).

trayectorias de un bloque a otro. Si un nodo solo tiene una arista de salida, esto se conoce como un bloque de "entrada", si un nodo solo tiene una arista de entrada, se conoce como un bloque de "salida". Por último, el análisis léxico convierte la sintaxis del código fuente en comodines de información en un intento de abstraer el código fuente y facilitar su manipulación.

Una de las principales ventajas de estas herramientas de identificación de errores, es que pueden ejecutarse con gran rapidez incluso en una base de código de gran tamaño y en programas parcialmente escritos. Así mismo, la principal desventaja de estas herramientas, es que no son efectivas a la hora de encontrar errores en tiempo de ejecución como los errores de referencia nula o de división por cero. Dado que su análisis es superficial, informan errores equívocos o falsos, técnicamente denominados "falsos positivos".

Los atributos de calidad pueden ser fácilmente categorizados por estas reglas, permitiendo identificar problemas si dichas reglas no se cumplen. Los atributos definidos por SpotBugs[34] son los siguientes:

- **Mala práctica (BAD_PRACTICE)**

Violaciones de la práctica de codificación recomendada y esencial. Los ejemplos incluyen código hash, código duplicado, excepciones no controladas, problemas de serialización y uso indebido de finalización.

- **Corrección (CORRECTNESS)**

Error probable: un error de codificación aparente que resulta ser un código que probablemente no era lo que pretendía el desarrollador.

- **Internacionalización (I18N)**

Errores de código que tienen que ver con la estandarización con respecto a la internacionalización y localización.

- **Vulnerabilidad del código malicioso (MALICIOUS_CODE)**

Código que es vulnerable a ataques o código no confiable.

- **Corrección de multiproceso (MT_CORRECTNESS)**

Errores de código que tienen que ver con hilos, bloqueos y memoria.

- **Rendimiento (PERFORMANCE)**

Código que no es necesariamente incorrecto, pero puede ser ineficiente.

- **Seguridad (SECURITY)**

[34] Morgenthaler, J. D., & Penix, J. (2008). software development tools Using Static Analysis to Find Bugs. Development.

El uso de información no confiable, de manera que pueda crear una vulnerabilidad de seguridad aprovechable de forma remota.

- **Código Evasivo (STYLE)**

Código que es confuso, anómalo o está escrito de una manera que conduce a errores. Los ejemplos incluyen variables locales inactivas, interrupciones de conmutación, conversiones no confirmadas y comprobación de valor nulo redundante.

7.2.3 Algoritmos de clasificación y predicción

En el aprendizaje automático y las estadísticas, la clasificación es un enfoque de aprendizaje supervisado en el que el programa informático aprende de la información que se le da y luego utiliza este aprendizaje para clasificar la nueva observación.

7.2.3.1 Naive Bayes

El modelo de clasificación de Naive Bayes se basa en el Teorema de Bayes [33], cuyo concepto es tener una probabilidad condicional que se basa en condiciones de acontecimientos de eventos, es decir busca la probabilidad que un evento Y suceda en el dado caso que un evento X ya haya sucedido, el modelo de Naive Bayes aplica este concepto en un modelo de aprendizaje supervisado en el que se provee de algoritmos de aprendizaje que se basan en datos previos[35].

Este espacio de muestras esta dividido en *n* partes tal que $\Omega = \{A_1, A_2, A_3, \ldots, A_n\}$, conociendo la probabilidad de cada una de las particiones A_i y la probabilidad de un suceso B que está condicionado a cada una de las particiones A_i, conocido como $P(B|A_i)$ se obtiene:

$$P(\mathrm{A}_i|\mathrm{B}) = \frac{P(\mathrm{A}_i \cap B)}{P(B)}$$

O dicha de otra manera

$$P(C_k|\mathrm{X}) = \frac{P(\mathrm{x}|C_k)\,P(C_k)}{P(x)}$$

Donde C representa cualquier hipótesis cuya probabilidad pueda verse afectada por los datos, P© es la estimación de la probabilidad de la hipótesis C antes de que se observe la información X. X corresponde a datos nuevos que no se usaron para calcular la probabilidad anterior, P(C|X), representa la probabilidad posterior, es la

[35] Serrano, A. (2017). Inteligencia artificial.

probabilidad de C dado X, es decir, después de que se observa X. P(X|C) es la probabilidad de observar X dado C. Como una función de X con C fija, indica la compatibilidad de la evidencia con la hipótesis dada. La función de probabilidad es una función de la evidencia(x), mientras que la probabilidad posterior es una función de la hipótesis(C).

Los clasificadores Bayes pueden ser entrenados en un ambiente de aprendizaje supervisado. En muchas aplicaciones prácticas, la estimación de parámetros para modelos ingenuos de Bayes utiliza el método de máxima verosimilitud; en otras palabras, puede trabajar con el modelo de ingenuidad de Bayes sin la necesidad de aceptar la probabilidad bayesiana.

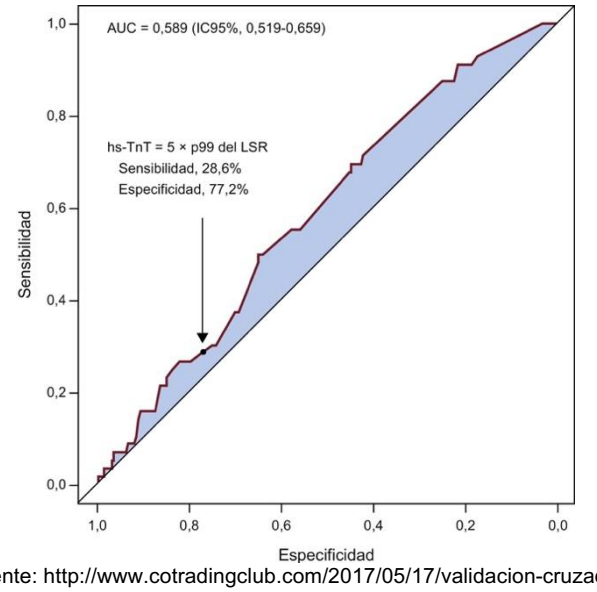
Una ventaja del clasificador de Bayes es que requiere solo una pequeña cantidad de datos para estimar los parámetros necesarios para la clasificación. Como se suponen las variables independientes, solo es necesario determinar las varianzas de las variables de cada clase y no la matriz de covarianza completa.

7.2.4 Característica operativa del receptor

Al evaluar el rendimiento de una prueba de detección, un algoritmo o modelo estadístico como la regresión logística, para la cual el resultado es dicotómico, se considera la sensibilidad, especificidad, valor predictivo positivo (VPP) y valor predictivo negativo (VPN). Sin embargo, tienen la desventaja de referirse a un punto único de corte y requieren una evaluación de equilibrio entre la sensibilidad y la especificidad, mientras que el VPP y VPN están influenciados por la prevalencia de la población. Las funciones operativas del receptor (ROC) ofrecen una representación gráfica del rango de posibles puntos de corte con su sensibilidad asociada frente a 1-especificidad, es decir, tasa de falsos positivos. Las estimaciones del área bajo la curva (AUC) incluyen una indicación de la utilidad del predictor y un medio para comparar dos o más modelos predictivos[36].

Figura 6 - Área Bajo la Curva (ROC)

[36] Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874.



Fuente: http://www.cotradingclub.com/2017/05/17/validacion-cruzada/

Las curvas ROC trazan la tasa (sensibilidad) positiva real, frente a la tasa de falsos positivos (1-especificidad) para los diferentes puntos de corte posibles de una prueba de diagnóstico. Cada punto en la curva ROC representa un par de sensibilidad / especificidad.

Al analizar el grafico ROC se deben tener en cuenta las siguientes reglas:

- Cuanto más cerca de la curva sigue el borde del lado izquierdo y el borde superior, más precisa es la prueba.
- Cuanto más cerca esté la curva de la diagonal de 45 grados, la precisión de la prueba disminuye.

8. ALCANCES Y LIMITACIONES

8.1 ALCANCE

El desarrollo de este proyecto propone un análisis de calidad dentro de un proceso de desarrollo convencional enfocado en las tendencias usadas en el mercado. Está dado por un experimento que permite acercar los métodos de aprendizaje automático, en una aplicación en fase de desarrollo a nivel exploratorio, es decir, la aplicación sobre la cual se ejecuta el método es netamente académica. Como base de conocimiento se toman las reglas de desarrollo básicas, explicadas a lo largo de este proyecto y se analizan a partir de fuentes existentes.

8.2 LIMITACIONES

8.2.1 Limitaciones metodológicas

Estas limitaciones están dadas a partir de la implementación de los algoritmos y la cantidad de estudios y trabajos previos a esta propuesta, los cuales a su vez son base del desarrollo del experimento con limitaciones en la factibilidad y precisión que tienen los algoritmos a implementar, puesto que al ser un tema que ha sido trabajado recientemente, la madurez de los algoritmos desarrollados está sesgado al error actual.

8.2.2 Limitaciones de fuentes (reglas)

Como se explicó previamente, los datos utilizados en este proyecto son tomados de bases y reglas existentes las cuales fueron descritas de manera preliminar. Conviene sin embargo advertir que existe la posibilidad de colisión entre ellas, razón por la cual se deben descartar las reglas que colisionen y se consideren irrelevantes por el uso actual en la industria.

8.2.3 Limitaciones de fuentes (datos de ejemplo)

Los datos de ejemplo son auto-informados o construidos manualmente, ya que los datos pueden ser insuficientes para contemplar todos los posibles escenarios, reiterando que las bases de datos y reglas que se tienen en cuenta son las de uso recurrente en la industria.

8.2.4 Limitaciones de ambiente de software

El ambiente de prueba está limitado al software en desarrollo de ejemplo. Este es el elemento de prueba para el experimento y está desarrollado en Java 1.7, por consiguiente, los experimentos están basados en dicho lenguaje.

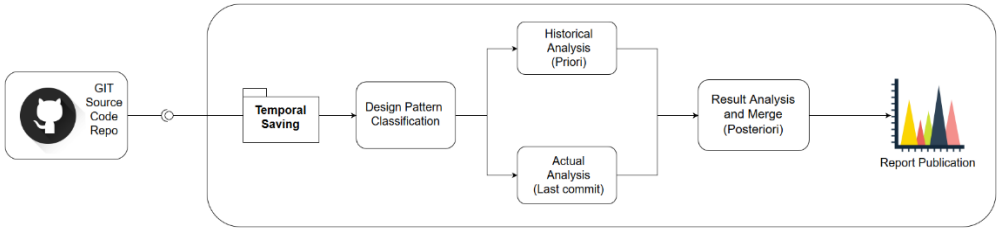
8.2.5 Limitaciones de software

El desarrollo del modelo se construye bajo el lenguaje Java 8 y Maven 3.3, como gestor de librerías, dependencia y compilación.

9. METODOLOGíA

El desarrollo de los objetivos se llevó a cabo mediante un proceso de reconocimiento en los patrones de diseño a través del modelo de *template*. Una vez los patrones de diseño fueron reconocidos y clasificados, se procede a realizar un análisis de código estático sobre todos los *commits* de la rama Master. Al mismo tiempo que se obtienen los resultados de los bugs en el análisis del código estático, se realiza una matriz de probabilidades clasificando los tipos de bugs, efectuando a su vez el cálculo de los bugs que se presentarán en el próximo commit, como es descrito en la Figura 7.

Figura 7 - Metodología



Fuente: Autor

9.1 OBTENCIÓN DEL CÓDIGO FUENTE

Para la obtención del código fuente de la aplicación al cual se le realiza el análisis, se efectúa un llamado al API de Git, el cual muestra por cada rama la cantidad de commits y sus respectivos identificadores, de tal manera que, por cada commit, se puede obtener el código fuente de la aplicación que se encuentra en análisis, siendo este descargado en la instancia actual de ejecución de la herramienta que realiza el análisis.

9.2 EXTRACCIÓN DE DATOS

Ya obtenido el código fuente de la aplicación sujeta a análisis, se realiza la extracción de las características de este para darle cumplimiento a GOF y POSA, extrayendo la información para ser evaluada por clases, métodos y atributos del código fuente, siendo extraído y compilado para que la clasificación y la predicción pueda llevarse a cabo.

9.3 DATOS DE REFERENCIA

El conjunto de datos son las reglas iniciales con el que el modelo de clasificación da inicio, por lo tanto, este debe ser un robusto conjunto de datos que permita al modelo clasificar los patrones de diseño.

El conjunto de reglas en las cuales se contemplan los patrones y anti-patrones de diseño especificados en POSA y GOF, son generadas de las buenas prácticas del desarrollo y arquitectura del software dándole así un marco, plantilla y esquema con el cual la clasificación es realizada.

Estas reglas están dadas por diferentes archivos conocidos como templates, los cuales definen la estructura básica de cada patrón mediante un sistema de lenguaje natural que permite establecer relaciones entre clases como herencia, declaraciones y abstracción. Estos templates establecen las reglas únicas a la hora de realizar la clasificación.

9.4 MODELO DE CLASIFICACIÓN DE PATRONES DE DISEÑOS

Una vez efectuados los procesos anteriores para la recolección de datos y estos hayan sido procesados, se define un modelo de aprendizaje automático teniendo en cuenta los algoritmos de clasificación planteados previamente. Basado en la implementación realizada por DP-Core[37], el conjunto de datos es adquirido y clasificado, mediante el proceso de template, los archivos base de clasificación permiten al algoritmo clasificar los patrones de diseño.

Este proceso es realizado mediante el análisis de relaciones que poseen las clases entre sí, todos los objetos de cada clase se extraen incluyendo el tipo de abstracción, seguido a esto se obtienen las conexiones entre las clases. Por ejemplo, si la clase A usa una conexión de A a B, dado que la clase A devuelve el tipo B, o para los casos de herencia donde la clase A extiende de la clase B.

9.5 MATRIZ DE EXCEPCIONES

Ya que la información fue clasificada y el código fuente fue descargado y compilado para cada uno de los commits, se crea una matriz de *n* x *m*, donde *n* representa los

[37] hemistoklis Diamantopoulos, Antonis Noutsos, and Andreas Symeonidis. "DP-CORE: A Design Pattern Detection Tool for Code Reuse." In Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD), pp. 160-169, 2016.

commits y *m* representa el atributo de calidad; para cada uno de los commits filtrados por los patrones de diseño detectados, se realiza un análisis de código estático, en el cual por cada commit y para cada atributo se realiza el conteo de las excepciones(bugs) encontradas y se disponen en la matriz, con el objetivo de tener el histórico de bugs durante el desarrollo del software que está siendo objeto de análisis.

9.6 MATRIZ DE PROBABILIDAD DE OCURRENCIA

Con la matriz de bugs completada, se procede a realizar el mapeo de los bugs que se encontraron, con el fin de hallar la probabilidad de ocurrencia del bug. Para esto se lleva a cabo un análisis de dicha matriz, mediante un proceso recurrente que analiza los bugs por cada commit y por cada atributo, que permita conocer la probabilidad de ocurrencia del bug en un momento específico.

Mediante el algoritmo Bayesiano la matriz de probabilidades es generada y para este caso particular, el espacio de muestreo es la matriz de bugs obtenida previamente, por consiguiente, cada uno de los atributos son analizados en el commit actual y todos los anteriores se examinan de forma recursiva. Teniendo en cuenta esto, el modelo bayesiano está dado por la ecuación 1, donde ϕ representa el atributo de calidad, ω representa los commits y β representa los bugs por atributo en un commit.

Ecuación 1 - Naive Bayes

$$P(\phi|\omega) = \frac{P(\omega|\phi) \cdot P(\phi)}{P(\omega)}$$

Este modelo se desglosa a través de la verosimilitud, la probabilidad a priori de la clase y el predictor de probabilidad a priori.

9.6.1 Verosimilitud

La verosimilitud representada como $P(\omega|\phi)P(\omega|\phi)$, está dada por la probabilidad conjunta, en la que para un commit y un atributo de calidad hay bugs presentes. Con el objetivo de saber la probabilidad de ocurrencia de un bug en un atributo de calidad dado un commit específico. Sabiendo que en el atributo de calidad hay una cantidad de bugs presentes $\beta_i|\beta_{i_1}$, siendo *i* el índice del atributo de calidad y *j* el

índice del commit; calculando el total de bugs en el commit, se obtiene $P(\omega|\phi)$ $P(\omega|\phi)$, mediante la división de β_{ij} β_{ij} y el total de bugs en el commit.
La ecuación resultante está expresada por:

Ecuación 2 - Verosimilitud

$$P(\omega|\phi) = \frac{\beta_{ij}}{\sum_{i=1}^{m} \beta_{ij}}$$

9.6.2 Probabilidad a priori de la clase

La probabilidad a Priori de la clase o simplemente llamado probabilidad priori $P(\phi)$ $P(\phi)$, define la probabilidad de ocurrencia de bugs dado un atributo de calidad, para este caso la ocurrencia ya no es en un bug específico, sino el total de bugs que se representa en el atributo. Este es calculado mediante la sumatoria de todos los bugs ocurridos en un atributo de calidad, dividido en el total de bugs de todo el proyecto de software que está siendo analizado. La representación matemática está expresada por:

Ecuación 3 - Probabilidad a Priori de la Clase

$$P(\omega) = \frac{\sum_{j=1}^{n} \beta_{ij}}{\sum_{j=1}^{n} \sum_{i=1}^{m} \beta_{ij}}$$

9.6.3 Predictor de probabilidad a priori

El predictor de probabilidad a priori $P(\omega)$ $P(\omega)$ está dado por la probabilidad del commit, haciendo referencia al commit como un momento en el tiempo, es decir, un evento ocurrido. Esta probabilidad es proporcionada por la sumatoria de bugs en un commit (el evento que sucedió) sobre el total de bugs. Lo que se quiere lograr es identificar la probabilidad de ocurrencia de un bug dado un evento sucedido. El predictor de probabilidad es representado como:

Ecuación 4 - Predictor de probabilidad a priori

$$P(\omega) = \frac{\sum_{i=1}^{m} \beta_{ij}}{\sum_{j=1}^{n} \sum_{i=1}^{m} \beta_{ij}}$$

La fórmula para obtener la matriz de probabilidades está dada por:

Ecuación 5 - Bayes Expandido

$$P(\phi|\omega) = \frac{P(\omega|\phi) \cdot P(\phi)}{P(\omega)} = \frac{\frac{\beta_{ij}}{\sum_{i=1}^{m}\beta_{ij}} \cdot \frac{\sum_{j=1}^{n}\beta_{ij}}{\sum_{j=1}^{n}\sum_{i=1}^{m}\beta_{ij}}}{\frac{\sum_{i=1}^{m}\beta_{ij}}{\sum_{j=1}^{n}\sum_{i=1}^{m}\beta_{ij}}}$$

Esta fórmula da como resultado la probabilidad de ocurrencia de un bug para un commit en un atributo de calidad, el cual genera una matriz de probabilidades que se distribuye a través del tiempo dados los atributos de calidad.

9.7 MODELO PREDICTIVO

Una vez obtenida la matriz de probabilidades, la cual indica cuales son las probabilidades de ocurrencia de un bug en un atributo de calidad, lo que se pretende es obtener la predicción de lo que sucederá con el software en commits futuros, y así determinar las acciones que se deben llevar a cabo. Para esto es necesario plantear una probabilidad con independencia condicional, la cual, siendo consecuente con el modelo de la matriz de probabilidades, se realiza el cálculo mediante el predictor de Naive Bayes, el cual es obtenido a través de la máxima probabilidad de un atributo de calidad multiplicado por la productoria de las probabilidades de ocurrencia del último commit y está representado por:

Ecuación 6 - Naive Bayes

$$P(\omega_{n+1}|\phi_1, \phi_2 \ \dots \ \phi_m) = \ argmax_{\omega}\, P(\omega_n) \prod_{i=1}^{m} P(\phi_i|\omega_n)$$

Una vez se obtiene esta probabilidad, cuyo resultado es un arreglo de probabilidades de dimensión *m* (cantidad de atributos de calidad), se puede calcular el valor numérico de los bugs que habrá en el siguiente commit, mediante la suma

de la probabilidad de ocurrencia del atributo de calidad en el último commit con la probabilidad Bayesiana de este. Finalmente, el resultado obtenido es multiplicado por el total de bugs en el atributo de calidad y su representación matemática se muestra a continuación:

Ecuación 7 - Predictor Numérico

$$\beta_{i\,n+1} = (P(\phi_i|\omega_n) + P(\omega_{n+1}|\phi_{\mathrm{i}}))\sum_{j=1}^{n}\beta_{ij}$$

10. DISEÑO METODOLÓGICO

A través de la herramienta Lugh se llevó a cabo el diseño del proyecto en desarrollo, ya que permite predecir la cantidad de bugs que ocurrirán en el próximo commit. Las funciones que permitieron la ejecución de este proyecto se describen a continuación.

10.1 CONSTRUCCIÓN DEL CONJUNTO DE DATOS

La obtención de datos se realiza por medio de Git, siendo necesario que el repositorio en GitHub sea público, conociendo de antemano el nombre del autor y el nombre de su proyecto. La información conseguida fue seleccionada basándose en las limitaciones mencionadas previamente.

La cantidad de clases en comparación con los atributos de la calidad, son elementales a la hora de realizar el análisis, es por esto que, de los repositorios seleccionados para el experimento, se selecciona repositorios que cumplan con un amplio histórico, una cantidad equitativa de patrones de diseño y una cantidad representativa de clases.

10.2 GENERACIÓN DE TEMPLATES DE PATRONES DE DISEÑO

Los templates generados para la detección de patrones de diseño, están establecidos de acuerdo con el método indicado por DP-CORE[38]. Estos registros están construidos en archivos de texto plano con extensión template, de tal manera que permiten describir las posibles relaciones que las clases tienen entre si, dependiendo del patrón de diseño que se quiera analizar. La estructura de los archivos está compuesta por nombre, miembros y relaciones.

- Nombre: Es la descripción del patrón de diseño, como por ejemplo Abstract Factory.

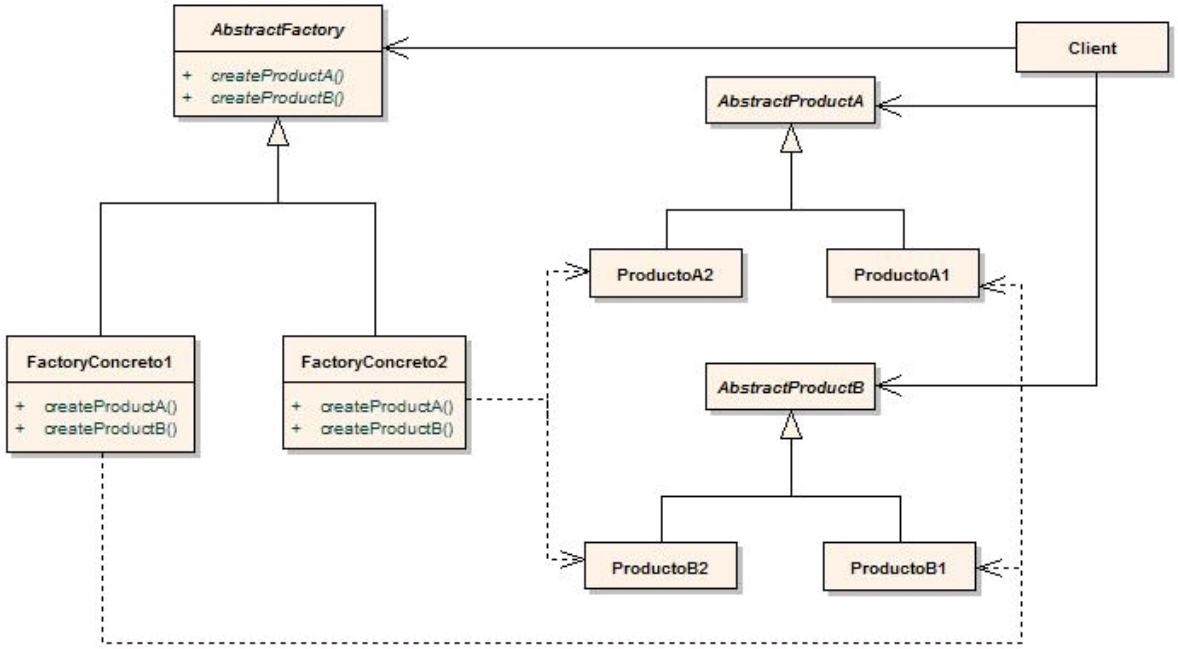
- Miembros: se definen los tipos de clases que se encuentran en el patrón de diseño y la descripción de este se realiza mediante un comodín que permite

[38] Themistoklis Diamantopoulos, Antonis Noutsos, and Andreas Symeonidis. "DP-CORE: A Design Pattern Detection Tool for Code Reuse." In Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD), pp. 160-169, 2016.

identificar la clase. Estos comodines aparecen en forma de caracteres alfabéticos, por orden alfabético y en mayúscula. Ejemplo: A, B, C. Junto a ellos se encuentra una breve descripción de la clase que permite identificar el rol que desempeña en el patrón de diseño.

- Relaciones: son descritas mediante el uso de comodines y las relaciones se establecen por palabras clave que el algoritmo de reconocimiento identifica para realizar la coincidencia de relación. Las palabras clave son: uses, inherits, creates, calls, references, y has; la relación entre clases describe cómo una clase se comporta con respecto a otra.

Figura 8 - UML Abstract Factory



Fuente: https://i0.wp.com/upload.wikimedia.org/wikipedia/commons/thumb/9/9d/Abstract_factory_UML.svg/677px-Abstract_factory_UML.svg.png

La Figura 8 muestra la interfaz AbstractFactory, que define los métodos implementados por AbstractProductA y AbstractProductB, mientras que ConcreteFactory1 y ConcreteFactory2 definen los métodos de los objetos del Producto. Se debe tener en cuenta que esta es una ilustración posible, debido a la facultad de tener un número diferente de ConcreteFactory, AbstractProduct o clases de productos. Por definición, un patrón de Abstract Factory debe incluir instancias de AbstractFactory, AbstractProduct, ConcreteFactory y Product. Los 4 miembros del patrón y sus conexiones se definen en la Figura 9.

Figura 9 - Template Abstract Factory

```
Abstract Factory
A Normal ConcreteFactory
B Abstracted AbstractFactory
C Normal Product
D Abstracted AbstractProduct
End_Members
A inherits B
C inherits D
A creates C
A uses D
End_Connections
```

Fuente: Autor

Por cada uno de los miembros del patrón, A, B, C y D, se observa su tipo de abstracción y su rol. El rol de un miembro es un campo de descripción que puede establecer el desarrollador, para que las instancias de patrón detectadas sean más completas. El patrón tiene cuatro conexiones, dos *inherits*, que son A (Concrete Factory) a B (Abstract Factory) y una de C (Product) a D (Abstract Product), un *creates* de A (Concrete Factory) a C (Product), y un *uses* de A (Concrete Factory) a D (Abstract Product).

10.3 IMPLEMENTACIÓN DE ALGORITMO DE CLASIFICACIÓN DE PATRONES DE DISEÑO

El algoritmo desarrollado recibe como entrada los objetos extraídos del proyecto examinado (y sus conexiones), así como el patrón a ser detectado, dado el formato definido en las subsecciones previas. Se itera sobre los objetos y se verifica si el objeto actual puede coincidir con el miembro del patrón actual. Esto se realiza llamando de forma recursiva a la función Detect y proporcionando el índice al miembro del patrón actual como el parámetro de profundidad d. En principio el algoritmo se inicializa con una profundidad igual a 0 (y candidato es un conjunto vacío). Iterando sobre el primer objeto, se verifica si su abstracción y sus conexiones son iguales con el miembro del patrón 0, si el objeto coincide con este miembro del patrón, entonces la función Detect se llama nuevamente pasando como parámetros los objetos y el candidato actualizado para que incluya el objeto, mientras que el parámetro de profundidad también se incrementa. Si en algún momento el objeto actual no coincide con el miembro del patrón actual, entonces la recursión se detiene. Cuando todos los miembros del patrón coinciden, el candidato se agrega a los candidatos del patrón detectado.

10.4 ANÁLISIS DE CÓDIGO ESTÁTICO

Mediante el uso de herramientas automatizadas como SpotBugs[39], los atributos de calidad son detectados mediante errores como: escribir "=" en lugar de "==" en las comprobaciones de condición. Se puede llegar a detectar fácilmente este "patrón de error" comprobando si el operador "=" se usa para la verificación de las condiciones. Esto se realiza de manera típica al hacer coincidir el patrón de codificación en el programa con el patrón de error esperado en el nivel AST (Abstact Sintax Tree). De esta manera se realizan diferentes procesos para poder obtener bugs mediante la ejecución de algoritmos.

10.4.1 Análisis de flujo de datos

En el análisis de flujo de datos (DFA) se recopila la información del tiempo de ejecución sobre los datos en los programas. Este análisis generalmente se realiza recorriendo el gráfico de control-flujo (CFG) del programa, de tal forma que puede considerarse como una representación abstracta de funciones en un programa o en una gráfica. Cada nodo en la gráfica representa un bloque básico y los bordes dirigidos se utilizan para representar saltos en el flujo de control.

Ahora bien, el DFA se puede ejecutar para encontrar errores como el acceso de puntero nulo, desde el punto en que se inicializa la variable del puntero y se quita la referencia, podemos encontrar las rutas en las que el valor de la variable del puntero aún es nulo cuando se le quita la referencia. El DFA puede ser intra-procedimiento o inter-procedimiento, es decir, el análisis puede limitarse solo a una función o al programa completo. El análisis se realiza normalmente mediante el uso de algoritmos estándar y no son intensivos en su computación.

10.4.2 Interpretación abstracta

La interpretación abstracta es aproximar la semántica del programa al reemplazar el dominio concreto de computación y sus operaciones, por un dominio abstracto de computación y sus operaciones. Para explicar la interpretación abstracta con un ejemplo, se considera la expresión (-123 * 456) donde se pretende saber cuál es el signo resultante de la operación sin tener que realizarla, sabiendo de antemano que la respuesta a esta expresión es un valor negativo de acuerdo con la ley de los signos. En otras palabras, la expresión se puede interpretar de manera abstracta como: (valor negativo * valor positivo) => valor negativo. Dicho de otra forma, si se aplican las operaciones aritméticas para encontrar el signo de la operación se está

[39] Spotbugs [en línea] Spotbugs [Citado el 10 noviembre, 2018] Disponible en internet: <https://spotbugs.github.io/>

realizando una interpretación concreta; Si por el contrario se abstrae y se realiza la aritmética para encontrar el signo, se está realizando una interpretación abstracta.

10.4.3 Control de modelos

La ejecución del programa se puede ver como la transición de un estado a otro del programa. La mayoría de los estados son válidos y algunos son estados de error. Un ejemplo de ello son los estados del programa cuando ocurre división por cero, interbloqueo o desbordamiento de búfer. La verificación de modelos es una técnica de verificación automatizada, donde el posible comportamiento del sistema (es decir, el comportamiento de implementación) se corresponde con el comportamiento deseado (propiedades especificadas). Dicho de otra manera, un verificador de modelos idealmente revisa todos los estados posibles del sistema y que las propiedades dadas se mantengan.

10.4.4 Consulta sobre el Programa

La idea fundamental detrás de la consulta del programa es que este pueda verse en datos estructurados (en otras palabras, una base de datos) que permitan realizar consultas para obtener la información deseada. Al implementar esta idea, el programa puede manipularse implícitamente como una base de datos, teniendo la ventaja de la facilidad de codificar bajo lenguajes de programación lógica como Java, los cuales utilizan una base de datos de hechos y las consultas en estos lenguajes permiten inferir relaciones del conjunto de hechos dado. Por esta razón, se utilizan ampliamente para el análisis estático y en particular para inferir patrones de diseño o anti-patrones.

10.5 FUNCIÓN ITERATIVA DE ANÁLISIS HISTÓRICO

Para poder realizar un histórico de todos los bugs en una aplicación que se encuentra en desarrollo, se realiza una iteración en los históricos de Git. Estos históricos están representados por los commits que ha sido publicados por los desarrolladores.

10.5.1 Lectura de históricos

Para realizar la lectura de históricos se utiliza la API de Git, la cual permite listar todos los commits que se encuentran en una rama, dado el proyecto que está siendo objeto de análisis obteniendo el total de commits y su identificador único. Estos tienen la particularidad de describir el autor, fecha y descripción de cada commit. Todos estos commits son considerados como “versiones” del código fuente del proyecto e indica los cambios que se han realizado durante el proyecto. Esta lectura

permite, además, tener un registro detallado de las funcionalidades nuevas, cambios hechos e incluso código eliminado.

Teniendo conocimiento del histórico y la referencia del identificador único del proyecto, el código fuente de cada commit es descargado, obteniendo así la información exacta de los datos al momento en el que fue publicado.

10.5.2 Iteración analítica

Esta iteración permite crear la matriz de bugs sabiendo la cantidad de commits. Para ello se realiza una iteración por cada uno de los commits existentes y como se mencionó previamente, se descarga el código fuente de ese commit con el objetivo de realizar la búsqueda de patrones de diseño y sobre estos realizar el análisis de código estático. El resultante es la matriz de bugs clasificada por los atributos de calidad para cada uno de los commits.

10.6 PREDICCIÓN DE BUGS

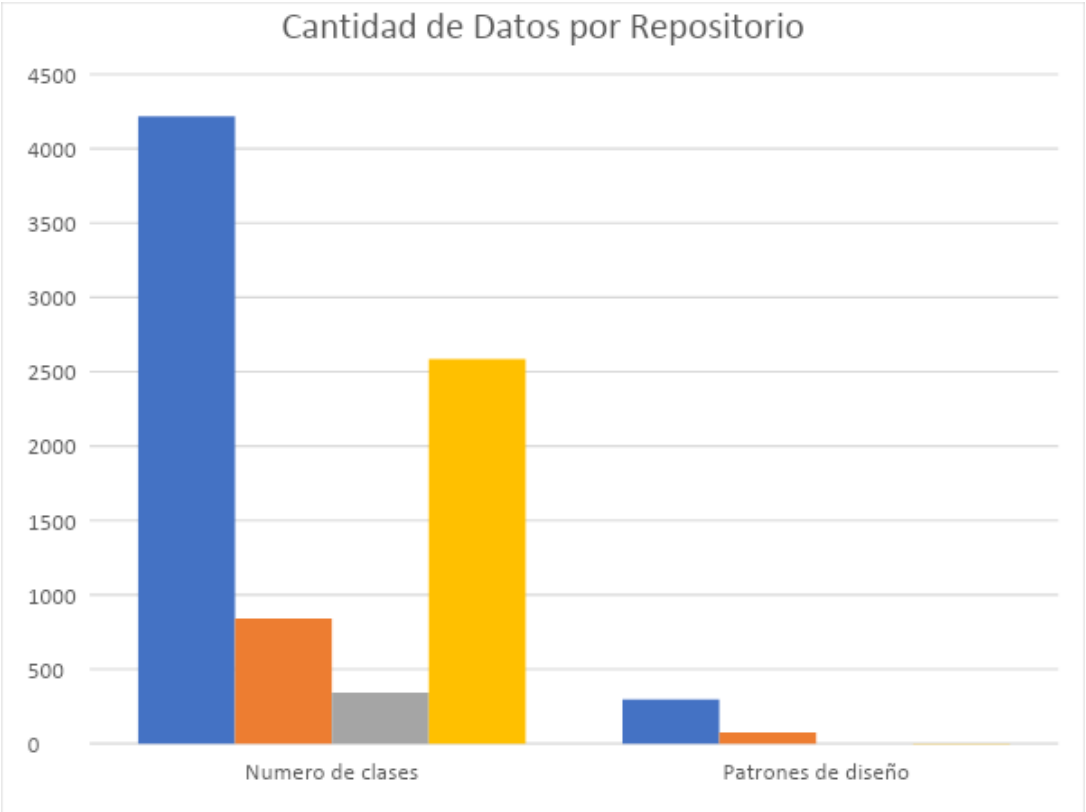
La principal herramienta de desarrollo utilizada en la ejecución de los algoritmos de este proyecto se llevó a cabo mediante el modelo bayesiano planteado anteriormente, a través del cual se pronostica y se precisa la cantidad de bugs que se verán reflejados en el próximo commit por medio del análisis matricial. Este proceso se llevó a cabo por medio de la clasificación probabilística en una línea de tiempo, detectando de manera significativa los errores que ocurren en la ejecución del software.

11. RESULTADOS

11.1 CONSTRUCCIÓN DEL CONJUNTO DE DATOS

Para la realización del experimento, el contenido seleccionado esta dado por la cantidad de clases, la cantidad de commits y la cantidad de patrones de diseño que se encuentren en el proyecto a ser analizado.

Figura 10 – Distribución de Datos en los Repositorios



Fuente: Autor

Tabla 1 - Repositorios de Git

Propietario	Repositorio	Numero de commits	Numero de clases	Patrones de diseño
saeidzebardast	java-design-patterns	10	4217	296
PushpinderSinghGrewal	lan-chat-app	20	840	75
Rohitjoshi9023	KBC--Kaun-Banega-Crorepati	22	342	0
AmbalviUsman	Billing-System	22	2584	2

Fuente: Autor

Dada esta distribución, se realiza la selección de los repositorios más propicios para realizar el análisis y la predicción de bugs, los proyectos seleccionados se pueden evidenciar en la tabla 1, en la cual se menciona el propietario del proyecto en Git, el nombre del repositorio, la cantidad de commits, el numero de clases en el proyecto y la cantidad de patrones de diseño. En la figura 10 se puede ver la relación que tiene cada repositorio con respecto a la cantidad de clases en comparación a la cantidad de patrones de diseño.

11.2 ESTRATEGIA DE CATEGORIZACIÓN Y PREDICCIÓN

Como resultado del modelo para la categorización y predicción, la herramienta Lugh es implementada (Anexo 1), la cual está en la capacidad de pronosticar la cantidad de bugs que potencialmente habrá en el próximo commit del código fuente. De manera automática la herramienta realiza el análisis histórico del código fuente mediante la implementación de los algoritmos mencionados previamente (10. Diseño metodológico) para así entregar como resultado el pronostico de los potenciales errores del proyecto analizado.

11.3 DESEMPEÑO DEL MODELO PREDICTIVO

Con el objetivo de determinar el nivel de desempeño del modelo predictivo y su comportamiento, una vez que el modelo de predicción arroja sus resultados se realiza una aproximación porcentual que permita indicar que tan precisa y exacta fue la predicción, para poder obtener estas medidas se recurre al uso de evaluaciones dadas por el Área Bajo la Curva y la Precisión.

11.3.1 Exactitud

La exactitud es definida por la cercanía del valor estimado con respecto al valor real, el cual es adquirido mediante el error que hay en las predicciones. Para obtener el valor de la exactitud se realiza el cálculo de uno menos el valor del error, como es descrito en la siguiente fórmula:

Ecuación 8 – Exactitud

$$Exactitud = 1 - \frac{|Valor\ Real - Valor\ Predcido|}{Valor\ Real}$$

En la tabla 2 y en la figura 11 se ven reflejados los porcentuales de exactitud para el total de bugs que arrojó la predicción.

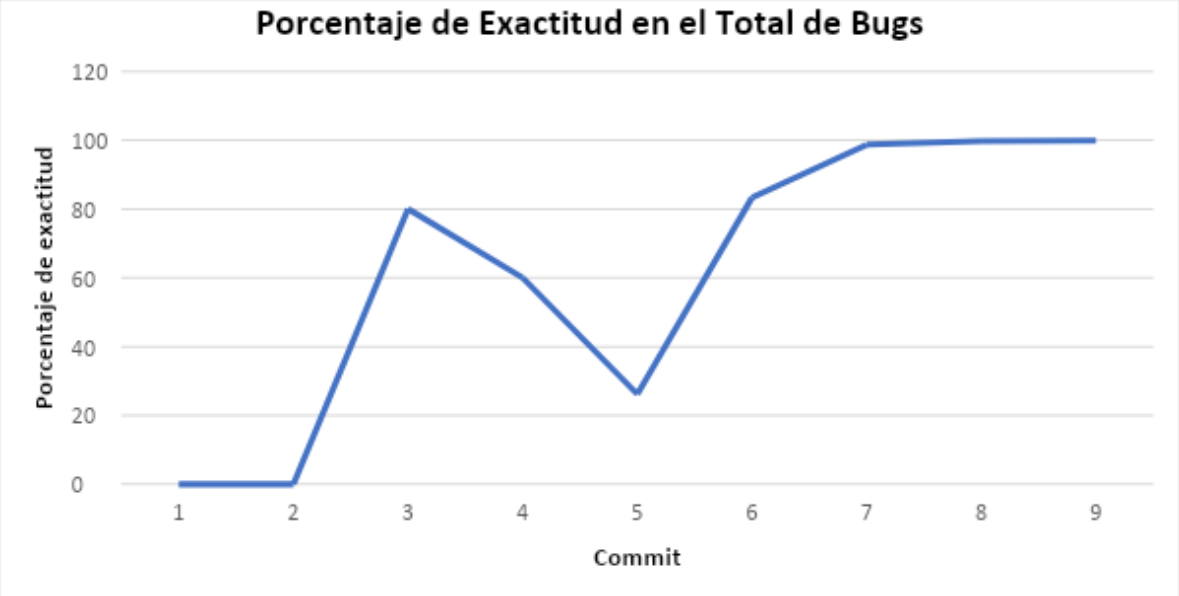
.

Tabla 2 – Porcentaje de Exactitud en el Total de Bugs

Porcentaje de Exactitud en el Total de Bugs	
Commit #	**% de Exactitud**
2	0
3	0
4	80
5	60
6	26.1640798
7	83.3303443
8	98.7076841
9	99.7815352
10	99.9401014

Fuente: Autor

Figura 11 – Porcentaje de Precisión en el Total de Bugs



Fuente: Autor

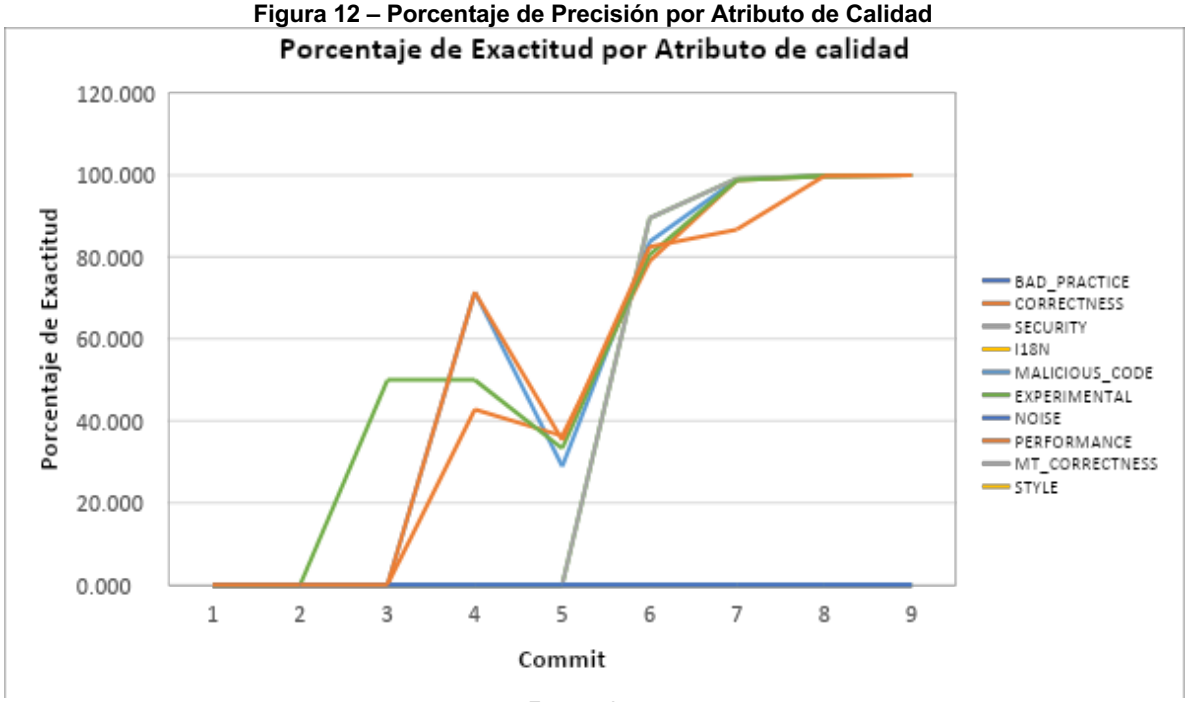
En la tabla 3 y la figura 12 se evidencian los porcentajes de exactitud por atributo de calidad que arrojó la predicción

Tabla 3 – Porcentaje de Exactitud por Atributo de Calidad

		Atributos de Calidad						
		BAD_PRACTICE	CORRECTNESS	SECURITY	I18N	MALICIOUS_CODE	PERFORMANCE	ST
Numero de Commit	2	0.000	0.000	0.000	0.000	0.000	0.000	
	3	0.000	0.000	0.000	0.000	0.000	0.000	
	4	0.000	0.000	0.000	0.000	0.000	50.000	
	5	71.429	0.000	0.000	42.857	0.000	50.000	7
	6	28.926	0.000	0.000	36.364	0.000	33.333	3
	7	83.750	89.485	0.000	78.971	89.485	80.588	8
	8	98.902	99.137	0.000	98.706	99.137	98.772	8
	9	99.780	99.814	0.000	99.752	99.814	99.762	9
	10	99.940	99.947	0.000	99.934	99.947	99.936	9

Fuente: Autor

Figura 12 – Porcentaje de Precisión por Atributo de Calidad

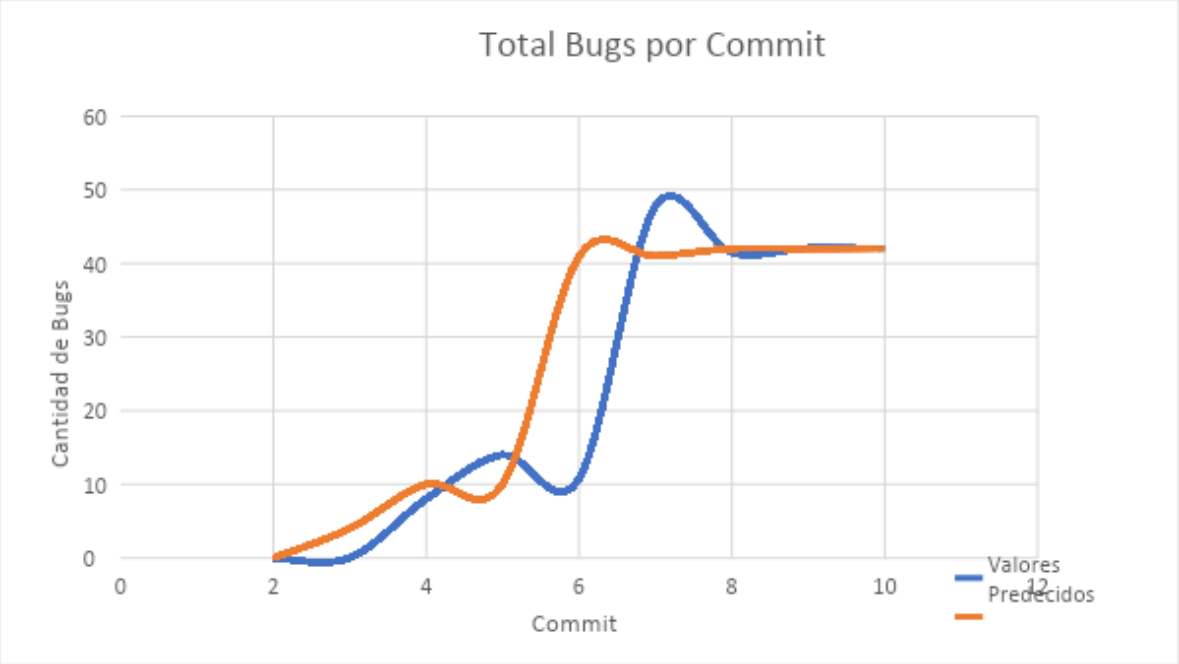


Fuente: Autor

Teniendo en cuenta las tablas 2 y 3, y las figuras 11 y 12, se evidencia la curva de tendencia incremental a medida que los commits avanzan, es decir la exactitud de la predicción es más acertada a medida que transcurre el tiempo, dando como resultado que la predicción realizada por el modelo converge a un 99% de exactitud, indicando que el valor pronosticado es acertado en comparación con el valor real.

Finalmente, en la figura 13 se efectúa la comparación entre los valores reales y los valores pronosticados. En la cual se puede observar que la línea roja (la cantidad de bugs real) comparado con la línea azul (la predicción de la cantidad de bugs) son asíntotas, llegando a acercarse cada vez más con el tiempo.

Figura 13 – Comparativo de Bugs Predichos y Real



Fuente: Autor

11.3.2 Precisión

La precisión es la medida en la cual se puede obtener la dispersión de los datos. Para poder demostrar el desempeño del modelo predictivo se realiza el análisis de la medida del área bajo la curva ROC, medidos mediante el uso de los atributos de sensibilidad y especificidad. Estos atributos están dados por las siguientes fórmulas:

Ecuación 9 – Sensibilidad

$$sensibilidad = \frac{VP}{VP + FN} = \frac{Valor\ Real}{Valor\ Real + (Valor\ Real - Predecido)}$$

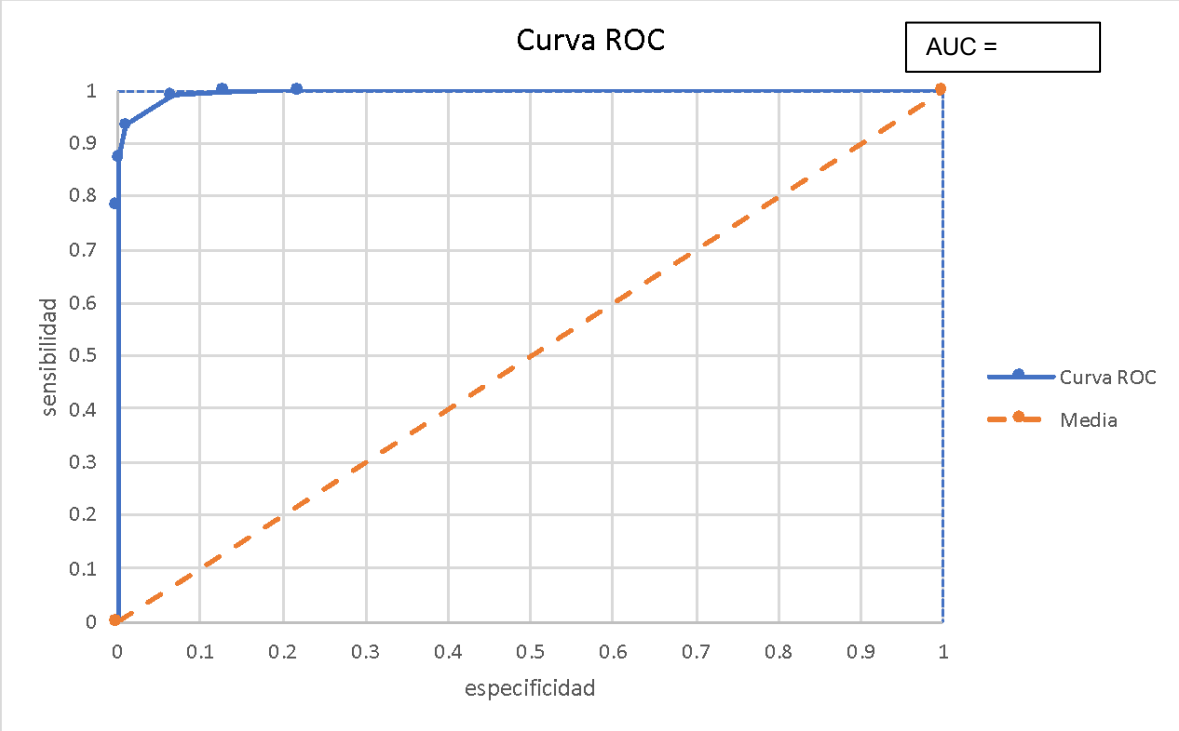
Ecuación 10 – Especificidad

$$especificidad = \frac{VN}{VN + FP} = \frac{(Valor\ Real - Predecido)}{Valor\ Predecido + (Valor\ Real - Predecido)}$$

Donde VP es verdadero positivo, VN es verdadero negativo, FN es falso negativo y FP es falso positivo. Teniendo en cuenta lo anterior, los valores pronosticados tomaron los porcentajes de predicción del modelo Bayesiano y como resultado se

obtuvo la curva ROC. La figura 14, muestra la tendencia cuyo objetivo debe ser el valor más cercano a 1.

Figura 14 – Curva ROC



Fuente: Autor

Una vez se calcula el área bajo la curva (AUC) el valor obtenido es 0.99624906 (muy cercano a 1), permitiendo considerar la afirmación de una predicción precisa, debido a que en las coordenadas (1,0) existe la máxima sensibilidad o, dicho de otra manera, una *Tasa de Verdaderos Positivos* de 99.6% en contraste a la Tasa de Falsos Positivos de 0.4%.

Estos resultados nos pueden confirmar la efectividad del modelo diseñado, si se tiene en cuenta que el porcentual del AUC indica que existe una tasa del 99.6% de precisión en la predicción.

12. CONCLUSIONES

Teniendo en cuenta los resultados, se puede evidenciar que, para la construcción de los datos, en el contexto del desarrollo de aplicaciones de software, hay una gran cantidad de proyectos con diferentes objetivos, pero con bases muy similares como la implementación de patrones de diseño y un método estándar de generación y compilación de clases. Adicionalmente se evidencia también que todos los proyectos, al hacer uso de herramientas como Git permite tener los históricos de los cambios en el código fuente mediante el manejo de los flujos de trabajo de GitFlow.

La ejecución de este proyecto permitió abrir una brecha en la cual, la inteligencia artificial juega un papel importante en la calidad de software, demostrándolo a través de la gran diversidad de proyectos existentes. El poder potencial que ejercen los modelos predictivos en el desarrollo de un software permite no solo conocer lo que sucederá con el software, sino también permitirá a los desarrolladores obtener alertas tempranas que permitan mejorar los procesos de manera continua. Dada esta premisa, herramientas automatizadas como Lugh, permite a los equipos de desarrollo enfocarse aún más en el desarrollo del código fuente y estar alerta a los cambios venideros reduciendo la necesidad de realizar inspecciones manuales.

Como se muestra en los resultados, la predicción que realiza en la herramienta es bastante óptima, como se puede evidenciar en el análisis de exactitud la predicción es más exacta a medida que más commits se hayan realizado, llegando a tener un porcentaje de exactitud bastante alto llegando a un valor cercano al 100%, dando a entender que la cantidad de bugs predichos con relación a la cantidad de bugs reales en un commit son parecidos, adicionalmente la precisión demostrada por la curva ROC da a entender que dada la sensibilidad y la especificidad del pronostico de bugs, la cantidad de verdaderos positivos es bastante alta, dando como resultado un porcentaje de 99.6% de precisión, Se puede concluir por lo tanto que el método de predicción usado muy bueno en términos de precisión y exactitud.

Sin embargo, al ser el desarrollo de software un proceso artístico, muchas características no pueden ser determinadas de forma metódica, ya que no hay un patrón único que permita definir cómo cada desarrollador en el mundo realiza su código o que nuevas estrategias tiene que aprender. Al fin y al cabo, los desarrolladores son artistas que crean una obra de arte en un lienzo en blanco.

13. TRABAJO FUTURO

Para trabajos futuros y referencias se recomiendan los siguientes puntos:

- Ampliar el análisis de datos a diferentes lenguajes de programación como C++, Scala, Python, entre otros, para comparar el desempeño de estos en ambientes característicos de Inteligencia Artificial.

- Realizar análisis que no estén sesgados únicamente a patrones de diseño, en gran medida porque surge la necesidad de aumentar análisis enfocados en los diferentes comportamientos estructurales que existen.

- Generar reportes en HTML y sistemas de alerta que le permita analizar al equipo de trabajo los reportes, estadísticas y resultados, en parte porque el uso de estas herramientas permite mitigar errores que afecten la operación de la industria que lo esté ejecutando, haciendo partícipes al equipo de trabajo en la toma de decisiones.

- Generar un ejecutable para poder ser integrado con herramientas de CI/CD como Jenkins: Este proceso se enfoca en lograr que cada uno de los eslabones de integración continua sean respetados y aprovechados al máximo, siendo este autónomo en su ejecución, permitiendo que la herramienta se ejecute automáticamente en fases tempranas contribuyendo con la toma de decisiones.

- Creación de técnicas de predicción con mayor velocidad y precisión: Los modelos de aprendizaje automático van en aumento, siendo a su vez mejorados continuamente dando cabida a la optimización de los algoritmos implementados y por ende a la herramienta en general.

- Generar un sistema de almacenamiento que permita a la aplicación tener los registros históricos de tal manera que no sea necesario realizar el análisis completo con cada ejecución: Al tener información considerable que constituya el crecimiento del proyecto que se va a analizar, conllevara un incremento de costos en los recursos con los que se disponga, razón por la cual almacenar los históricos permite a la herramienta analizar solo la última versión.

14. INSTALACIONES Y EQUIPO REQUERIDO

14.1 HARDWARE

- Computador portátil u ordenador con ratón, teclado, monitor, requisitos mínimos de procesamiento, almacenamiento, memoria, dispositivo receptor de wifi o conexión local.
- Celular.
- Conexión de Internet.

14.2 SOFTWARE

- Sistema operativo Windows 10, Ubuntu 16 o MacOS 19.
- IntelliJ Idea: IDE que permite generar proyectos en Python y Java.
- Java SE: Ambiente del sistema que permite desarrollar, compilar y ejecutar proyectos JAVA.
- Maven: Gestor de paquetes de JAVA.

15. ESTRATEGIAS DE COMUNICACIÓN Y DIVULGACIÓN

• **POSTER**: un póster es un medio visual para comunicar los resultados de trabajos, experiencias, proyectos de investigación, etc., y se puede dar a conocer de diferentes maneras; su presentación en congresos, conferencias, mesas redondas, comunicaciones orales es uno de ellos. Esta forma de exposición contribuye al intercambio de información entre los asistentes a los eventos, quienes tienen la oportunidad de interactuar directamente con los autores y obtener información adicional si están interesados.

• **TRABAJO DE GRADO**: El Proyecto de Trabajo de Grado es la presentación escrita de la planificación y ejecución de los pasos seguidos en el desarrollo del Trabajo de Grado.

16. BIBLIOGRAFIA

¿Qué es un Patrón de Diseño? [en línea] [Citado el 24 mayo, 2018] Disponible en internet: <https://msdn.microsoft.com/es-es/library/bb972240.aspx>

A successful Git branching model [en línea] Vincent Driessen. [Citado el 16 abril, 2019] Disponible en internet: < https://nvie.com/posts/a-successful-git-branching-model/ >

Análisis de Código [en línea] s.l.: Analyzing+Source+Code. [Citado el 24 mayo, 2018] Disponible en internet: <https://docs.sonarqube.org/display/SONAR/Analyzing+Source+Code>

Balaji, S. (2012). Waterfall vs v-model vs agile: A comparative study on SDLC. WATERFALL Vs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC, 2(1), 26–30.

Bayes, R. T., Laplace, P., & Jeffreys, S. H. (2017). Bayes theorem, 1–11.

Branching strategies with tfvc [en línea] microsoft. [Citado el 18 febrero, 2019] Disponible en internet: <https://docs.microsoft.com/en-us/vsts/tfvc/branching-strategies-with-tfvc?view=vsts>

Breiman, L.: Random forests. Mach. Learn. 45(1), 5–32 (2001)

Cohn, M. (2009). The Forgotten Layer of the Test Automation Pyramid.

Continuous Integration [en línea] Martin Fowler. [Citado el 5 junio, 2018] Disponi-ble en internet: <http://martinfowler.com/articles/originalContinuousIntegration.html>

Dagpinar, M., Jahnke, J. H., & Canada, B. C. (2003). Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison

Dai, W., & Ji, W. (2014). A MapReduce Implementation of C4. 5 Decision Tree Algorithm, 7(1), 49–60.

Detection Tool for Code Reuse." In Proceedings of the Sixth International Sympo-sium on Business

Dustin, E., Rashka, J., & Paul, J. (2017). Automated software testing, (April), 1–37.

Fawcett, T. (2006). An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874.

From, E. (2007). Design Patterns, 368–386.

Hauser, T. (2009). the Art of Unit Testing. Library Journal (Vol. 128).

hemistoklis Diamantopoulos, Antonis Noutsos, and Andreas Symeonidis. "DP-CORE: A Design Pattern Detection Tool for Code Reuse." In Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD), pp. 160-169, 2016.

International Software Testing Qualifications Board. (2011). Certified Tester Foundation Level Syllabus, 85.

Java design patterns [en línea] Saeid Zebardast. [Citado el 16 septiembre, 2018] Disponible en internet: <https://github.com/saeidzebardast/java-design-patterns>

Jeskanen, M. (2015). Non-functional testing: security and performance testing, (November).

Kenett, R., & Baker, E. (2010). Process Improvement and CMMI® for Systems and Software.

Li, M., Li, H., Zhou, Z.H(2009). Semi-supervised document retrieval.

Li, M., Zhang, H., Wu, R., & Zhou, Z. H. (2012). Sample-based software defect prediction with active and semi-supervised learning.

Louridas, P. (2006). Static code analysis. IEEE Software, 23(4), 58–61.

MISRA C: 2012 Technical Corrigendum 1 Technical clarification of British Library Cataloguing in Publication Data. (2017), (June).

Modeling and Software Design (BMSD), pp. 160-169, 2016.

Morgenthaler, J. D., & Penix, J. (2008). software development tools Using Static Analysis to Find Bugs. Development.

Myers, G. J., Badgett, T., & Sandler, C. (2011). Software Testing Tutorial, 2–7.

Practical test pyramid [en línea] Martin Fowler. [Citado el 5 junio, 2018] Disponi-ble en internet: <https://martinfowler.com/articles/practical-test-pyramid.html>

Rajagopalan, S. (2014). REVIEW OF THE MYTHS ON ORIGINAL SOFTWARE DEVELOPMENT MODEL, 5(6), 103–111.

Ruparelia, N. B. (2010). Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3), 8.

Scrum Values [en línea] s.l: scrum-values. [Citado el 24 mayo, 2018] Disponible en internet: <https://www.scrumalliance.org/learn-about-scrum/scrum-values>

Serrano, A. (2017). Inteligencia artificial.

Song, Q., Jia, Z., Shepperd, M., Ying, S., & Liu, J. (2011). A general software defect-proneness prediction framework.

Spotbugs [en línea] Spotbugs [Citado el 10 noviembre, 2018] Disponible en in-ternet: <https://spotbugs.github.io/>

Static Code Analysis [en línea] OWASP [Citado el 21 marzo, 2019] Disponible en internet: <https://www.owasp.org/index.php/Static_Code_Analysis>

Supporting agile manifesto principle 6: face-to-face contact [en línea] Platinum Edge. [Citado el 24 mayo, 2018] Disponible en internet: <https://platinumedge.com/blog/supporting-agile-manifesto-principle-6-face-face-contact>

Themistoklis Diamantopoulos, Antonis Noutsos, and Andreas Symeonidis. "DP-CORE: A Design Pattern

Tuteja, M., & Dubey, G. (2012). A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models, (3), 251–257.

Why agile development races ahead of traditional testing. [en línea] computerweekly. [Citado el 24 mayo, 2018] Disponible en internet: <https://www.computerweekly.com/feature/Why-agile-development-races-ahead-of-traditional-testing>

17. ANEXOS

Anexo A: Conjunto de datos. Se entrega el conjunto de datos con los cuales se ejecutaron las pruebas de el método de predicción

URL: https://github.com/saeidzebardast/java-design-patterns
URL: https://github.com/PushpinderSinghGrewal/lan-chat-app
URL: https://github.com/Rohitjoshi9023/KBC--Kaun-Banega-Crorepati
URL: https://github.com/AmbalviUsman/Billing-System

Anexo B: Repositorio. Se entrega el código utilizado del experimento realizado, para fines educativos e investigativos, bajo licencia libre MIT (MIT, X11).

URL: https://github.com/jpduque/lugh